

Harnessing Hibernate

精通



Hibernate

O'REILLY®



机械工业出版社
China Machine Press



James Elliott, Tim O'Brien
& *Ryan Fowler* 著
刘平利 译

O'Reilly精品图书系列

精通Hibernate

Harnessing Hibernate

[美]艾里特 (Elliott,J.) 著

刘平利 译

ISBN: 978-7-111-26487-3

本书纸版由机械工业出版社于2009年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.bbbvip.com

新浪微博 @研发书局

腾讯微博 @yanfabook

目 录

O'Reilly Media, Inc.介绍

译者序

前言

第一部分 Hibernate快速入门

第1章 安装和设置

获得Ant发布版本

检查Java版本

获得Maven Tasks for Ant

安装Maven Tasks for Ant

使用HSQLDB数据库引擎

获得Hibernate Core

建立项目层次结构

第2章 映射简介

编写映射文档

生成Java类

编制数据库Schema

第3章 驾驭Hibernate

配置Hibernate

创建持久化对象

检索持久化对象

建立查询的更好方法

第4章 集合与关联

集合的映射

集合的持久化

集合的检索

使用双向关联

使用简单集合

第5章 更复杂的关联

关联的主动加载和延迟加载

有序集合

扩充集合中的关联

关联的生命周期

自身关联

第6章 自定义值类型

用户自定义类型

定义一个持久化的枚举类型

使用自定义的类型映射

使用持久化的枚举对象

建立组合自定义类型

第7章 映射标注

Hibernate标注

为模型对象添加标注

另一种方法

第8章 条件查询

使用简单条件查询

组合式条件查询

投影和聚合的条件查询

在关联中应用条件查询

示例查询

面向属性的Criteria工厂

第9章 浅谈HQL

编写HQL查询

选择属性和其他部件

排序

使用聚合值

编写原生SQL查询

第二部分 与其他工具的集成

第10章 将Hibernate连接到MySQL

建立MySQL数据库

连接到MySQL

尝试一下

查询数据

第11章 Hibernate与Eclipse: Hibernate Tools使用实战

在Eclipse中安装Hibernate Tools

创建一个Hibernate控制台配置

更多的编辑支持

Hibernate Console视图

代码生成

映射图表

第12章 Maven进阶

什么是Maven

安装Maven

项目的构建、测试以及运行

使用Maven生成IDE项目文件

用Maven生成报告

Maven项目对象模型

Maven构建的生命周期

使用Maven Hibernate3插件

超越Maven

第13章 Spring入门: Hibernate与Spring

Spring是什么

编写数据访问对象

创建应用程序上下文对象

把所有组件装配在一起

第14章 画龙点睛：用Stripes集成Spring和Hibernate

安装Stripes

准备Tomcat

创建Web应用程序

增加Stripes

处理关联

附录A Hibernate类型

附录B Criteria API

附录C Hibernate SQL方言

附录D Spring事务支持

附录E 参考资源

作者简介

封面介绍

O'Reilly Media, Inc. 介绍

为了满足读者对网络 and 软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权机械工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为 20 世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的 Web 服务器软件），O'Reilly Media, Inc. 一直处于 Internet 发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠

他们及时地推出图书。因为O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc.知道市场上真正需要什么图书。

译者序

基本上所有应用程序都要与数据打交道，如何操纵和处理底层数据库曾经是一个让人非常头痛的问题，尤其对于Java新手来说，更是无从下手。

如果直接使用最底层的JDBC来访问数据库，再在代码中夹杂上无数的SQL语句，以这样的方式来手工编写代码不仅单调乏味、易于出错，而且会占用整个应用程序的很大一部分开发工作量。关键是这样得到的最终产品往往与底层的数据库紧密地耦合在一起，如果要更换数据库，必须花费大量的人力资源。

优秀的面向对象开发人员厌倦了这种重复性劳动，他们开始采用通常的“积极”偷懒做法，即创建工具，使整个过程自动化。对于关系数据库来说，这种努力的最大成果就是对象/关系映射（ORM）工具，而Hibernate则是这些工具中的典型代表。

Hibernate是一个免费的开源Java包，它使得与关系数据库打交道变得十分轻松，就像数据库中包含的是普通Java对象一样，不必考虑如何把它们从神秘的数据库表中取出（或放回数据库表中）。

Hibernate解放了广大Java程序开发人员，使他们可以专注于应用程序的对象和功能，而不必担心如何保存它们或稍后如何找到它们。

Hibernate之所以能够流行，应该归功于以下优点：

1) Hibernate是JDBC的轻量级对象封装，它是一个独立的对象持久层框架，与App Server、EJB没有什么必然的联系。Hibernate可以在任何JDBC可以使用的场合。

2) Hibernate是一个和JDBC密切关联的框架，所以Hibernate的兼容性与JDBC驱动、数据库都有一定的关系，但是与使用它的Java程序、底层数据库没有任何关系，也不存在兼容性问题。

3) Eclipse等主流Java集成开发环境对Hibernate有很好的支持，在大型项目，特别是持久层关系映射很复杂的情况下，Hibernate效率非常高。

为了让以前对Hibernate了解不多的Java爱好者快速掌握Hibernate的基本配置、使用方法、经验技巧，以及它与其他常用开发工具的协同配合，本书的作者由一个简单而现实的示例入手，从数据表的创建，讲到各种基于数据库的操作，甚至还创建了一个简洁的Web网站，内容涉及Hibernate的方方面面。讲解非常细致，不仅包括了足以帮助读者理解的源代码，而且对于每一操作步骤，作者都给出了详细的操作命令。相信读者在阅读和实践本书示例的过程中一定不会遇到太大的问题，而且能够以最短的时间来掌握Hibernate，这应该这就是本书最可贵的价值所在了。

本书在结构上分为两大部分。前一部分主要介绍Hibernate框架自身的功能，后一部分则介绍Hibernate与其他IDE和开发工具的配合使用。所有讲解并非照本宣科式地照搬API文档和参考手册，而是时时处处渗透着作者在使用Hibernate过程中所领悟到的经验和体会，尤其是在讲解Hibernate的关联映射配置时，虽然我自诩已经使用Hibernate很多年了，但还是学到不少知识点，这些在API和参考手册中没有遇到和使用过。第二部分中介绍的各种开发工具也是成熟的Java开发人员不可或缺的利器，对它们的掌握和理解，是超越普通程序员的必经之路。

在翻译过程中，虽然我力求在忠于原文的基础上，尽可能从专业Java开发人员的角度来做到信、达、雅，但由于自身水平有限，必定会有诸多不足，希望各位读者不吝指教。

感谢华章公司陈冀康编辑的理解和支持；也感谢我的朋友孙凤萍、董彦奇、聂磊、高原等对本书翻译和校对工作的大力帮助。另外还要感谢我的家人，没有他们的支持也无法完成这本书的翻译。

最后，祝大家能够在阅读中享受技术进步带来的乐趣！

刘平利

2008年12月1日

前言

Hibernate是为Java设计的轻量级对象/关系映射（object/relational mapping）服务。这是什么意思？这就是说，Hibernate可以让你用普通的Java对象的形式来简洁而有效地处理关系数据库中的信息。不过，这样的说明仍然无法贴切地表达这项技术是多么有用和令人激动。持有这种观点的人并非只是我一个：Hibernate 2.1赢得了《Software Development》杂志第14届“框架库和组件”震撼大奖（Jolt Award）。

（本书是《Hibernate: A Developer's Notebook》的后续更新版本，我非常荣幸地编写了这本书。这本书第1版本介绍的是Hibernate 2，它获得了第15届Jolt技术类图书生产力大奖（Productivity Winner）。

那么，Hibernate到底神奇在哪里呢？所有非凡的应用程序（甚至许多平凡的应用程序）都需要存储和使用各种信息，也就都会涉及关系型数据库的使用。与Java对象世界不同，数据库通常要求使用者具备一定的技巧和专业知识。如何连通这两个世界曾经是一段时期内的一项重要任务，但这也是一件非常复杂而乏味的工作。

大多数人要事先写出一些非常繁琐的SQL语句，再将这些语句作为字符串嵌入到Java代码中，接着再使用JDBC（Java database connectivity）执行查询语句和处理结果。JDBC已经发展成为一个与数据库通信的、功能丰富且非常灵活的程序库，虽然现在基于这种方法

还可以提供一些简化和改进的措施，但在Java中使用JDBC还相当繁琐。对于大量的数据处理，我们需要某种功能更强大的工具，将对数据库的查询从代码中分离出去，并以面向对象的方式将它们组件化，以简化对数据库的操作。

多年来，我自己开发的软件中就使用了这样的轻量级（甚至是超轻量级）对象/关系映射层功能组件。该组件最初起源于我的同事Eric Knapp为Lands' End电子商务网站开发的Java数据库连接和查询池缓存系统。这个系统引入了外部SQL模板的思想，可以通过名称来访问模板，并有效地与运行时数据组合起来，以生成实际的数据库查询。只是它后来才支持在模板中增加一些简单的映射指令，将这些模板直接绑定到Java对象。

虽然它远不及像现在的Hibernate这样有强大的功能和系统，但这种方法在很多不同规模的项目和各种环境中已证实具有很大的价值。直到本书的第1版，我们一直都在使用这种方法，在为Cisco公司的CallManager平台建立IP电话应用程序时，我们最后采用了这种方法。不过，现在再做新项目时，我们会改用Hibernate。在学习完本书以后，你会明白为什么要做这样的选择，而且也可能会做出同样的决定。Hibernate会为你做很多事情，简单到让你几乎忘记是在处理数据库。需要什么对象，就直接拿来使用即可。这就是这种技术的优点和使用方式。

你可能会问，**Hibernate**和**Enterprise JavaBeans (EJB)**有什么关系？它们是彼此竞争的技术吗？在什么情况下应该使用哪种？事实上，你可以同时使用这两种技术。但是并非每个应用程序都需要**EJB**的复杂性，多数应用程序只需要使用**Hibernate**直接与数据库交互，就足够了。另一方面，对于非常复杂的三层（**three-tier**）应用程序环境而言，**EJB**有时是不可或缺的。在这种情况下，**EJB Session bean**可以使用**Hibernate**来持久保存数据，或者也可以用于持久化**BMP**实体bean。

事实上，**EJB**委员会深受**Hibernate**的影响，最终接受了**Hibernate**的"plain old Java objects"（**POJO**）的方式进行持久化处理，这是一种功能强大、使用方便的持久化方式，并在**EJB 3**中引入了**Java Persistence Architecture (JPA)**（可以脱离**EJB**环境使用）。**Hibernate 3**其实也以一种完全可移植的方式实现了**JPA**（不过，在第7章中可以看到，你可能仍旧希望使用**Hibernate**的**JPA**扩展）。

Hibernate的开发很明显已经成为**Java**和关系型数据库交互的分水岭事件。**Java**界应该感谢**Hibernate**之父**Gavin King**和他的团队所做出的贡献，让我们的开发更加简单些吧。这本书就是要帮助你尽快地掌握这项技术。

本书怎么使用

本书最初是O'Reilly的Developer's Notebook系列的一部分，可帮助读者快速掌握有用的最新技术。虽然本书扩展了很多Hibernate用户可能想要了解的技术，但本书不打算成为Hibernate的完整参考手册。本书反映了作者对该系统研究的成果，从最初的下载，到项目的配置，通过一系列项目演示了如何完成各种实践目标。

阅读示例并实践一下，你不但能够快速地搭建好Hibernate环境，并且可以立即将它用于实际项目的开发。这就好像你“跟着我”走过我绘制的一片领地，沿途中，我会指出有用的路标和危险的陷阱。

虽然我一定会介绍一些背景知识，解释Hibernate的工作原理和原因，但这总是针对某项任务。有时，我会建议你参阅一些参考文档或其他在线资源，以便深入了解一些底层的概念或其他Hibernate使用方式的相关细节。

在读过前面几章之后，就不需要按照章节顺序依次阅读了，可以直接跳转到你特别感兴趣或关注的主题。你可以自己构建示例代码，也可以从本书的网站下载完整的源代码（可以在前一章示例代码的基础上，按照当前章节的说明，自己动手修改代码，来实现正在阅读的代码示例）。如果你正在学习的示例和前面的示例有关，同时你也有兴趣了解，则可以随时跳转到先前的示例。

本书排版字体约定

本书的字体有特定的约定，提前了解这些约定将有助于你对本书的理解。

斜体 (*Italic*)

用于文件名、文件扩展名、URL、应用程序名称、强调以及第一次引入的新术语。

等宽字 (`Constant width`)

用于Java类的名称、方法、变量、属性、数据类型、数据库元素以及以文本方式出现的代码片段。

等宽黑体字 (`Constant width bold`)

用于在命令行输入的命令，以及突出演示在运行示例中插入的新代码。

等宽斜体字 (`Constant width italic`)

用于说明输出结果。

本书网站

本书的网站地址是：

<http://www.oreilly.com/catalog/9780596517724/>，提供了一些你想要了解

的重要信息。本书所有示例按章节组织，都可以在以上网站中找到。

这些示例文件已经压缩成ZIP和TAR文件。

多数情况下，各章节都会用到一些相同的文件，随着示例功能的不同，这些文件不断增加一些新代码。下载文档中每一章的目录都是相应示例系统的状态快照（snapshot），反映了该章的所有变动和新增内容。

如何联系我们

请将有关本书的评论和问题寄送给出版商：

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）奥
莱利技术咨询（北京）有限公司

O'Reilly的每一本书都有专属网站，你可以在那找到关于本书的相关信息，包括勘误表、示例代码以及其他的信息。本书的网站地址是：

<http://www.oreilly.com/catalog/9780596517724/>

询问技术问题或对本书进行评论，请发电子邮件到：

bookquestions@oreilly.com

欲获取有关我们的图书、会议、资源中心（Resource Center）以及O'Reilly Network的更多信息，可以访问我们的网站：

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

致谢

首先非常感谢Gavin King、Christian Bauer、Steve Ebersole、Max Andersen以及其他参与创建和改进Hibernate的人。Hibernate这个工具让Java开发人员的工作变得如此简单，我们对它投入了巨大的热情，这也就是为什么在Jim编写了第1版本之后，我们三位作者还要决定继续扩展和更新这本书的原因。希望它确实如此！

我们要特别感谢我们的技术审阅者Michael Podrazik、Stefan Winz、Henri Yandell，感谢他们耐心地提出细致的、有帮助的建议。有了他们的贡献，这本书才会变得更好。

我们也要感谢O'Reilly生产部的Keith Fahlgren、Adam Witwer以及其他成员，他们帮助我们尽快掌握了新的DocBook XML编辑环境，也让这本书看起来更加漂亮。

Stripes示例包括了Tim Fennell开发的代码，请在Apache Software License（apache软件许可）下使用。Stripes使用许可协议位于下载代码的ch14/stripes_license.txt中。

James Elliott

不管怎么样，都要先感谢我的父母，虽然我们原来住在乡下，他们还是排除万难，培养我对计算机领域的兴趣。也要感谢我的搭档Joe，他忍受着已经深深着魔的我。此外，也要感谢我的老板Berbee，给我深入研究Java的机会，并积累起作为一个可复用API设计的架构师所需要技巧；并且，让我不必担心程序设计世界中难以理清的专利权以及平台特定问题的琐碎事务；给了我那么多了不起的同事；还有，当我因为编写这本书而需要利用这些经验时，他给了我很大的支持。

当我需要在第2版中扩展一些相关的技术，例如Spring和Stripes时，Ryan和Tim给了我帮助。他们的热情和专业知识的让这个拖延很

久的版本得以进展的主要因素。

Marc Loy邀请我协助《Java Swing》第2版的工作，让我和O'Reilly那些优秀的家伙取得联系，Mike Loukides从那时起和我共事也很有耐心，甚至鼓励我自己写一本书。他发现Hibernate就是我起步的最佳主题，后来的事实也证明这是非常正确的决定，所以我再次扩展这本书。《Java Swing》那本书的校对编辑Deb Cameron扮演着极为关键的角色，让我把犹豫不决的写作冲动化为受益良多的实际行动。我也感谢她愿意把我“借出去”，不用再协助《Learning Emacs》第3版的工作，这让我可以专注于Hibernate的写作计划。

我还要感谢本书第1版的两位技术审阅者：Adrian Kellor和Curt Pederson。他们看过很早期的草稿，帮助我确定了创作的基调和方向，而且还强化了我对这项写作计划的价值的热情。当本书的第1版成型时，Bruce Tate通过实际使用和开课讲授Hibernate，对本书做了很重要的检查，提出了相当不错的建议和鼓励。Eric Knapp以科技类大学教科书的角度审查了本书大部分的内容，并提醒我应该脚踏实地。Tim Cartwright在最后阶段加入，看到的基本是最后完整的草稿，他认为Hibernate是一个在未来极具潜力的平台，并对内容和表述方式提供了大量和相当有用的回馈。

Ryan Fowler

我要感谢Jim邀请我合作编写这本书，也感谢他在我的创作和日常工作中对我的指导。我也要感谢Tim提供的技术帮助和对我的宽慰。感谢Mike Loukides耐心地为我指点创作的方向。我也得感谢我的妻子Sarah，感谢她对我的帮助、耐心和爱。没有她的支持，事情一定会变得很糟糕。最后我还要感谢我的父母，是他们给了我成就现在的事业和发展的工具。

Timothy O'Brien

感谢Jim和Ryan邀请我编写本书有关Spring和Maven的章节。感谢Mike Loukides提供了创作和合作的良好环境。Keith Fahlgren为写作提供的支持，他简直就是无价的资源宝库。O'Reilly的出版技术团队也为我们解决各种DocBook标记内容和版本控制问题，并花费了他们很多宝贵时间。

感谢Stefan Winz、Robert Sorkin、Ahmed Abu-Zayedeh、Bob Hartlaub、Rock Podrazik以及Jeff Leeman，他们为Spring和Maven相关章节的代码示例提供了必要的基础资料和无私的测试。也感谢我的女儿Josephine Ann，她为Hibernate标注相关的章节提供了重要的反馈信息；Josephine虽然只有两岁多，但是她很快地就学了一些Hibernate；感谢她抽出观看《Blue's Clues》（[\[1\]](#)）的时间，帮我发现了我的pom.xml文件中一些幼稚的错误。最后要感谢我的妻子Susan，她非常优秀。

[1] 一部成功的美国幼儿电视动画片。

第一部分 Hibernate快速入门

我们的第一个目标就是尽快地了解Hibernate的最新进展。这一部分的大多数章是基于《Hibernate: A Developer's Notebook》

(O'Reilly) 的内容并进行了更新，反映了Hibernate 3带来的主要变化。示例代码现在使用的都是一些开发工具的最新版本，我们通过它们来提供一个方便而且实用的Hibernate开发环境。还有一章专门介绍Java 5的标注功能，使用标注，而不是用XML映射文件，也可以配置Hibernate映射。

注意：当然，和其他任何关于活跃的开源项目的图书一样，图书介绍的内容总赶不上开源项目的发展！附录E列出了本书讨论的工具的特定版本以及应对变化的指导思想。

因为我们采用Maven来帮助下载许多工具和库，所以使用这本书来着手学习和实践书中的代码示例会更加容易。因为我们希望你可以明白，没有什么理由不去亲自实践一下这些代码示例。

在熟悉了Hibernate的基础之后，第二部分将演示如何将Hibernate绑定到其他组件环境，让各种组件互相配合，以发挥更大的作用。

万变不离其宗，现在就开始学习吧。

第1章 安装和设置

我一直很惊讶，竟然会有这么多免费而又好用的开源Java工具。多年前，我开发一个JSP的电子商务项目时，需要一个轻量级对象/关系数据库映射服务，那时还没有Hibernate这样的工具，只能自己构建了一个这样的组件。这个组件经过几年的发展，开发出一些很酷、很独特的功能。但是在我发现了Hibernate以后，我想在下一个项目中，就不会再继续使用自己熟悉的那个系统了（我当然对自己的系统抱有偏爱），而是会使用Hibernate。用过之后，你一定会知道Hibernate有多棒！

正在读这本书的你，一定急于想知道这种功能强大而且使用方便的技术，是如何架起连接Java对象和关系数据库这两个世界之间的桥梁的！Hibernate很好地充当了这个角色，它并不很复杂，所以学习起来也不困难。为了展示这一点，本章将要指导你理解Hibernate的用法，让你看看为什么Hibernate会这么令人激动。

之后的章节将介绍在更复杂环境（例如Spring和Stripes）下，把Hibernate作为它们的组成部分的应用，以及它和其他数据库的配合使用。第1章的目标是要向你展示，使用Hibernate构建一个基本的、自我包含的环境，并且用它完成真正的操作是多么容易的。

获得Ant发布版本

可能有些令人意外，在运行Hibernate之前需要做几件与Hibernate本身无关的事。首先，你必须搭建一个开发环境，以供示例代码可以运行。也可以为你可能构建的任何实际项目奠定坚实的基础，这将是令人高兴的意外收获。

如果在你的Java项目中，还没有使用Ant去管理构建（build）、测试（test）、运行（run）以及打包（package）的工作，现在就是开始使用Ant的好时机。本书的示例都是Ant驱动的，所以，你得安装一个能用的Ant才能运行示例代码，并验证在系统中对代码做出的修改，这才是最佳的学习方式。

首先，获得Ant的二进制发布版本，并安装它。

为何在意

我们选择使用Apache Ant来处理示例有几个原因。Ant很方便，而且功能强大，它已经成为基于Java开发的标准构建工具，而且是免费、跨平台的工具。如果使用Ant，我们的示例将可以在任何Java环境中一样地正常运行；也就是说，本书的任何读者都不会因为运行示例而遇到麻烦。这也意味着，我们可以少花一点力气就能够做很多很酷

的事情，尤其是几个**Hibernate**工具特意支持**Ant**之后，更是如此。我们会教你如何利用这些工具（值得注意的是，最近更复杂的**Java**项目经常使用的是**Maven**（[\[1\]](#)），它增加了很多其他的项目管理功能。所以我必须从二者中挑选一个，本着尽可能简单和实用的原则，我就决定继续使用**Ant**来管理这些示例）。如果你目前正在使用**Maven**作为代码构建工具，你会注意到我们使用**Maven**的**Ant**任务（**Task**）来管理**Ant**构建的依赖关系。虽然**Maven**的发展势头强劲，但**Ant**仍然是目前**Java**开发中使用最广泛的构建工具。每一章的示例代码文件夹中也包含一个**Maven**的Maven进行编译。在许多情况下，使用**Maven Hibernate3**插件，**Maven**构建文件提供的功能与**Ant**的build.xml文件一样。第12章介绍了如何用完整的**Maven**来构建和部署**Hibernate**应用程序的方法，但本书大部分示例仍旧使用**Ant**作为构建工具，同时使用**Maven Ant Task**来查找和下载需要的各种库文件，包括库文件之间互相信赖的文件。

为了能够使用这些功能，需要做的第一件事就是先安装**Ant**，让它可以正常运行起来。

注意：我以前觉得奇怪，可以用**Make**，为什么还要用**Ant**？现在，我已经明白用**Ant**来管理**Java**的代码构建有多么美妙，没有**Ant**还真的不行。

应该怎么做

Ant的二进制发布包可以在<http://ant.apache.org/bindownload.cgi>下载。滚动网页，找到Ant的当前最新版本，然后下载适合的压缩文件格式。选择一个适合存放的位置保存文件，然后解压。压缩文件展开的目录就是ANT_HOME。假如你把压缩文件解压到目录/usr/local/apache-ant-1.7.0，你可能会想创建一个符号链接（symbolic link）以方便使用，同时当你升级到新版本时，也可以免去更新环境配置的麻烦：

```
/usr/local%ln-s apache-ant-1.7.0 ant
```

安装好Ant之后，需要做一些设置才能让它正常工作。你得将Ant的bin目录（在这个例子中，就是/usr/local/ant/bin）添加到命令路径中。还需要设置环境变量ANT_HOME，将其设定为安装Ant的最顶级目录（在这个例子中，就是/usr/local/ant）。至于如何在不同的操作系统中执行以上这些处理步骤，如果需要的话，可以参阅Ant的手册（<http://ant.apache.org/manual/>）。

[1] <http://maven.apache.org/>.

检查Java版本

当然，我们假定你已经安装好了Java software development kit (SDK)。目前你应该使用Java 5或更新的版本，因为新版本的SDK会提供一些有用的新功能。尽可能使用最新稳定版本的SDK, Java 5或Java 6都可以支持本书的所有示例。用Java 1.3也可以使用Hibernate2的大部分功能，但你得用1.3版本的Java编译器重新构建Hibernate JAR文件。发布版本越新，对Java版本的要求就越高；而且Java 5已经发布很长时间了，它本身就提供了很多优点，所以我们没有必要为兼容早期的JDK而花费时间。我们的示例都假定你用的至少是Java 5，如果使用更低版本的JDK，那么就得做大量修改调整。运行以下命令可以查看JDK版本：

```
%java-version
java version"1.6.0_02"
Java (TM) SE Runtime Environment (build 1.6.0_02-b06)
Java HotSpot (TM) Client VM (build 1.6.0_02-b06, mixed mode,
sharing)
```

你也应该使用官方发布的Java版本（例如Sun或Apple发布的版本）。在编写本书时，我们的技术审阅者发现GNU公共授权的“功能类似”的Java实现并不能正确运行这些工具和示例代码。许多Linux发布版本安装的默认Java环境就是GNU的。如果你正在使用Linux，可能

需要自己下载Sun的JDK，并确保使用的是正确的版本（通过运行`java-version`命令）。既然Sun已经开放了Java的源代码，希望将来这种情况会得到改善，到时候可能在任何自由软件版本中都会自带Sun JRE和JDK。不过，在那一天实现以前，你必须自己下载。

在编写本书时，基于Debian的发布版本可以用它们的安装管理工具来安装Sun JDK（Ubuntu的"Feisty Fawn"和"Gutsy Gibbon"发行版本就自带了JDK 5和6）。Red Hat系列的发布版本仍然需要直接从Sun Microsystems的网站下载Java。具体情况具体分析吧。

安装好以后，就应该能够启动Ant进行测试，来确认一切都没有问题：

```
%ant-version
Apache Ant version 1.7.0 compiled on December 13 2006
```

发生了什么事

嗯，目前还不多，不过现在已经可以尝试我们稍后将要提供的示例了，再以这些示例作为起点，去做实际的Hibernate项目。

如果你是Ant新手，最好先简单阅读一下它的手册来了解Ant的工作原理和功能。这样可以让你了解示例中用到的`build.xml`文件是怎么回事。如果你开始或已经喜欢上了Ant，想深入研究，那么你可以仔细

阅读它的手册（[\[1\]](http://ant.apache.org/manual/)），或阅读O'Reilly的《Ant: The Definitive Guide》（当然，应该先把这本书看完）。

其他

Eclipse（[\[2\]](http://www.eclipse.org/)）、JBuilder（[\[3\]](http://www.borland.com/jbuilder/)）、NetBeans[\[4\]](http://www.netbeans.org/)，还是其他Java IDE？嗯，你当然可以使用这些IDE，但是怎么把Ant整合到IDE的构建过程中，就是你自己的事了（有好几种IDE已经支持Ant，所以你可能已经走在前面了；对于其他IDE，你可能还需要跨越学习的障碍）。如果都行不通，你还可以使用IDE开发自己的程序代码，然后使用我们提供的一个build脚本，从命令行来调用Ant。

如果你使用的是Maven，则可以通过在任意一章的示例目录或最顶级的examples目录中执行`mvn eclipse: eclipse`，来生成Eclipse IDE项目文件。如果在examples目录中运行`mvn eclipse: eclipse`，Maven将为每一章的示例生成一个Eclipse项目。第12章将详细介绍如何用Maven来构建示例代码，第11章将详细介绍Hibernate的Eclipse工具的用法。

[\[1\]](http://ant.apache.org/manual/) <http://ant.apache.org/manual/>.

[\[2\]](http://www.eclipse.org/) <http://www.eclipse.org/>.

[\[3\]](http://www.borland.com/jbuilder/) <http://www.borland.com/jbuilder/>.

[\[4\]](http://www.netbeans.org/) <http://www.netbeans.org/>.

获得Maven Tasks for Ant

稍等一下，难道我还没有讲完用Ant构建本书示例项目的用法？确实还没有说完，不过，这也并不是全部。虽然本书以Ant作为示例构建的基础，我们决定本书应该通过Maven Tasks for Ant来演示一下Maven优秀的依赖（dependency）管理功能。本书的最初版本在这一部分只用了几页的篇幅来介绍如何下载和管理一系列第三方库：包括从Jakarta Commons Lang到CGLIB。如果这些工作由你亲自来做，最少也得花费你宝贵的几分钟的时间，而且说明如何操作和操作本身费事又繁琐。本书中，我们在build.xml文件中声明了项目需要依赖的库文件，并由Maven负责下载和管理这些依赖文件。这样就节省了许多步骤和时间。好，现在就开始安装Maven Tasks for Ant。

应该怎么做

有两种方法来整合Maven Tasks for Ant：第一种方法是将需要的JAR文件放在Ant的lib目录，第二种方法就是在Ant的构建（build）文件中用typedef声明来包括antlib定义。我们打算使用前一种方法，将maven-ant-tasks-2.0.8.jar文件添加到Ant的lib目录中，因为这样对示例的build.xml文件修改量最少。首先，从Maven的网站^[1]下载需要的JAR文

件，在它的首页上应该可以看到用于下载Maven Tasks for Ant的链接（如图1-1所示）。



图 1-1 Maven网站上Maven Tasks for Ant的下载链接

在编写本书时，Maven Tasks for Ant的版本是2.0.8。点击Maven Tasks for Ant 2.0.8的链接，选择一个镜像，就可以下载到一个名为maven-ant-tasks-2.0.8.jar的文件。将该文件保存到本地目录。

[1] <http://maven.apache.org/>.

安装Maven Tasks for Ant

接下来，将下载好的maven-ant-tasks-2.0.8.jar文件复制到ANT_HOME/lib目录中。如果你从头至尾按本章的说明来操作，现在应该已经下载并安装好了Ant。还应该设置一个名为ANT_HOME的环境变量，当然，你也要了解Ant的安装目录是什么。在将maven-ant-tasks-2.0.8.jar复制到ANT_HOME/lib目录以后，任何build.xml文件都可以包含相应的命名空间（namespace），以使用Maven Tasks for Ant。

如果你运行示例使用的计算机上有多个用户，而且你没有将JAR文件放到ANT_HOME/lib目录的管理权限，不必担心，你也可以将maven-ant-tasks-2.0.8.jar文件放在~/.ant/lib这个目录中。Ant也会自动在这个目录中查找任何JAR文件。

在将maven-ant-tasks-2.0.8.jar文件复制到ANT_HOME/lib目录以后，你应该能够运行以下命令，以检查当前UNIX的类路径（class path）中是否包含了maven-ant-tasks-2.0.8.jar：

```
%ant-diagnostics|grep maven|grep bytes  
maven-ant-tasks-2. 0.8.jar (960232 bytes)
```

在Windows中，运行ant-diagnostics，并检查输出的类路径包括的类库列表中是否包含了maven-ant-tasks-2.0.8.jar。

使用HSQLDB数据库引擎

Hibernate支持非常多的关系数据库，可能对于下一个项目中你打算使用的关系数据库，**Hibernate**就已经提供了支持。我们需要为示例挑选一个数据库，幸运的是，眼前就有一个好的选择。这是一个免费的、纯Java的开源自由软件项目，它功能强大，足以担当我们的商业软件项目中的数据存储支持系统。令人惊讶的是，**HSQLDB**也相当完善而且安装简单（简单到我们可以让**Maven**在新版中负责它的安装），所以在此讨论它很恰当（你是否听说过**Hypersonic-SQL**，现在叫做**HSQLDB**。**Hibernate**的很多说明文档还沿用旧的名称）。

如果你偶然访问<http://hsqldb.sourceforge.net/>，发现这个项目好像已经停止了，不要惊慌。这是错误的网址，它是**HSQLDB**项目的前身。图1-2演示了该项目当前的主页，它还相当活跃。

为何在意

本书示例是基于数据库的，每个人都可以下载这些示例，方便地在此基础上做试验，其间不需要转换任何SQL方言（**dialect**）或操作系统命令，就可以和你的数据库一起使用（也许这意味着你可以节省下一两天的时间，不用去研究怎么下载、安装和配置某种常用数据库环

境)。最后，如果你是HSQLDB新手，那么在用过之后感到印象深刻和好奇心十足的概率肯定很高，最终在你自己的项目中会选择使用这种数据库，正如其项目主页上所说的：

HSQLDB是用Java编写的领先的SQL关系数据库引擎。它提供了一个JDBC驱动程序，支持ANSI-92 SQL的一个丰富子集（BNF树格式），以及SQL 99和2003增强（enhancement）。它提供了一个小巧（Applet版本的体积小于100kB）而快速的数据库引擎，以及基于内存和磁盘的两种数据表，支持嵌入式和服务器模式。此外，还包括了一些工具，例如微型Web服务器、内存查询（in-memory query）和管理工具（能够作为Applet运行），以及很多演示例子。

应该怎么做

当构建本书的示例时，Maven Ant Tasks将自动从位于<http://repo1.maven.org/maven2/>的Maven仓库（repository）下载HSQLDB JAR（以及其他需要的JAR）。所以，如果你想马上体验一下，可以直接转到1.7节。否则，如果你想下载HSQLDB供自己使用，或者想查阅它的文档、在线论坛或邮件列表，就可以访问它的项目主页<http://hsqldb.org/>。点击下载当前最新稳定版本（latest stable version）的链接（在编写本书时，最新的版本是1.8.0.7，如图1-2所示）。这将打开一个典型的SourceForge下载页面，上面列出了当前选中的版本可以

提供的下载链接。选择一个合适的镜像站点，就可以开始下载ZIP文件。

注意：去吧，下载HSQLDB。哎呀，它们的个头都很小！



图 1-2 HSQLDB主页上最新的稳定版本的链接

其他

Hibernate对其他数据库的支持怎么样呢?不用担心，Hibernate现在可以支持MySQL、PostgreSQL、Oracle、DB2、Sybase、Informix、Apache Derby等各种数据库（本书将在第10章和附录C中教你如何为不同的数据库指定各自的“方言”（dialects））。不过，如果你真的需

要，可以试着从一开始就去搞清楚怎么和你最喜欢的数据库打交道。这也意味着你在跟着本书示例走时多花费一些额外工夫，同时也会错过发现HSQLDB美妙之处的大好机会。

获得Hibernate Core

不需要再讲什么激励的话吧！你选择这本书的目的就是想学习如何使用Hibernate。或许并不太令人感到意外，Hibernate中为应用程序提供对象/关系映射服务的核心部分称为Hibernate Core。当构建本书的示例时，Maven会自动为你下载Hibernate和它的所有相关的依赖文件。即便新版的随书代码示例可以通过Maven Ant Tasks来获取Hibernate，你也可以亲自下载最新的Hibernate发布版本，浏览它的源代码，或只是查看在线文档、论坛或其他支持资源。如果你已经准备好了Hibernate，则可以跳过这一节，直接学习1.7节的内容。

应该怎么做

先访问Hibernate的主页<http://hibernate.org/>，要获得其完整的发布版本，需要找到"Download"链接，如图1-3的左边所示。

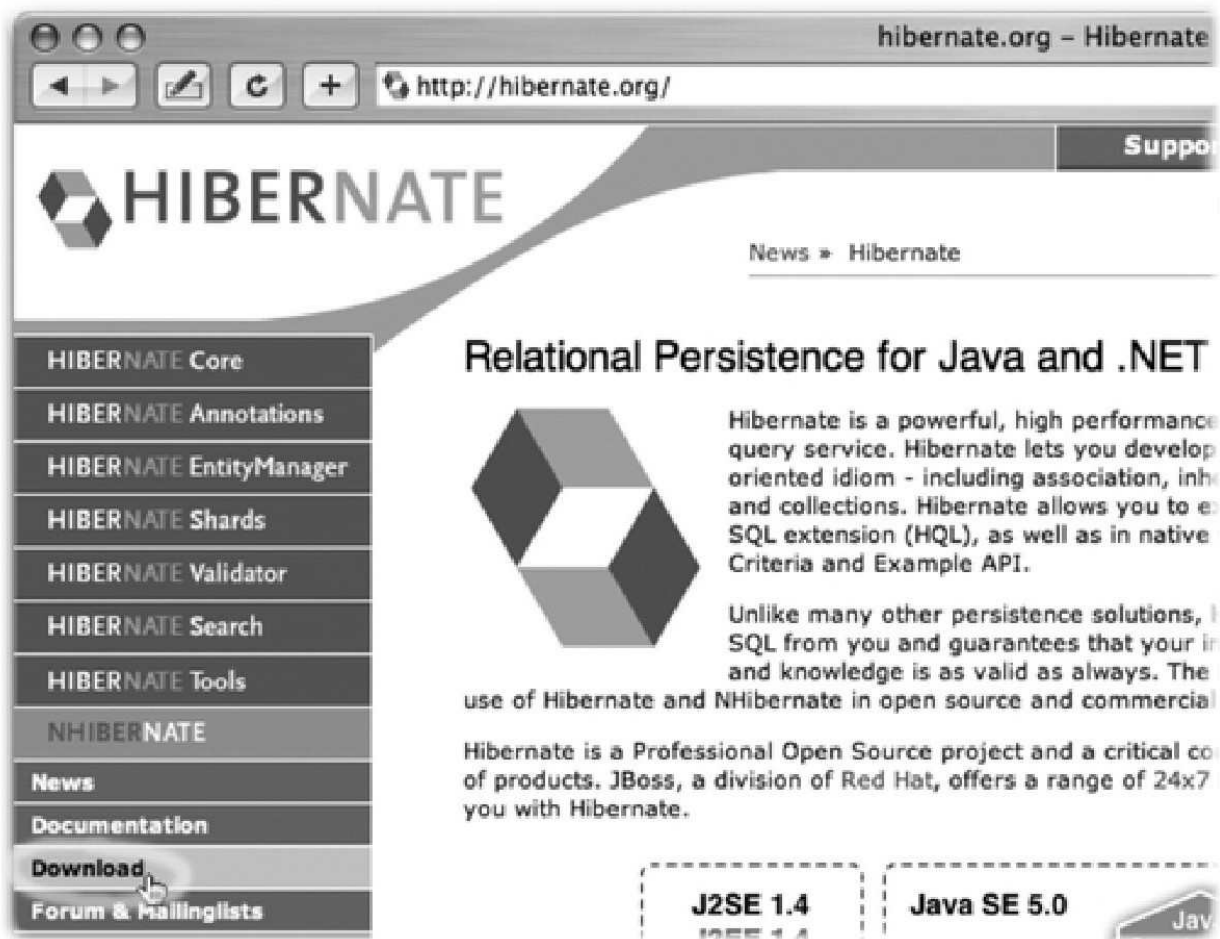


图 1-3 Hibernate主页上的Download链接

页面上的"Binary Releases"部分将列出Hibernate Core的推荐下载的版本（如果你有足够的勇气，可以尝试下载“开发”（Development）发布版本，但是最安全的还是支持下载最新的“产品”（Production）发布版本）。选择好以后，点击表格相应行上的"Download"链接（如图1-4所示）。

Binary Releases

Package	Version	Release date	Category	
Hibernate Core	3.2.5.ga	31.07.2007	Production	Download
Hibernate Annotations	3.3.0 GA	20.03.2007	Production	Download
Hibernate EntityManager	3.3.1 GA	29.03.2007	Production	Download
Hibernate Validator	3.0.0 GA	20.03.2007	Production	Download
Hibernate Search	3.0.0 Beta4	1.08.2007	Development	Download
Hibernate Shards	3.0.0 Beta2	02.08.2007	Development	Download
Hibernate Tools	3.2.0 Beta9	13.01.2007	Development	Download
NHibernate	1.2.0.GA	03.05.2007	Production	Download
NHibernate Extensions	1.0.4	24.01.2007	Production	Download
JBoss Seam	1.2.0 Patch1	28.02.2007	Production	Download

[Browse all Hibernate downloads](#), [Browse all NHibernate downloads](#)

图 1-4 Hibernate二进制发布版本

之后会打开一个SourceForge的下载页面，包含了你刚才选择下载的版本，以及可供选择的文档格式。选择对你最方便的压缩文件格式进行下载。下载的文件名看起来就像hibernate-3.x.y.tar.gz或hibernate-3.x.y.zip这样（在编写本书时，文件名开始以hibernate-3.2.5.ga命名，因为Hibernate 3.2.5的第一个稳定版本就是当前的产品发布版本）。

选择一个适合的地方将文件解压。

在Hibernate的下载页面，也可以看到"Hibernate Tools"部分（Download链接将打开一个名为"JBoss Tools"的页面，不过仍然可以在

上面找到**Hibernate Tools**)。它们提供了几个有用的功能, 这些功能虽然不是使用**Hibernate**的应用程序所必须的, 但是对开发人员创建某些应用程序来说非常有帮助。我们稍后首次对**Hibernate**进行试验时, 将使用其中的一个工具来生成**Java**代码。这个工具的文件名看起来应该类似于**hibernatetools-3.x.y.zip**的样子(不一定和**Hibernate**本身的版本号一样, 通常可以使用的只有**beta**版本; 在**Hibernate**的下载页面中, 位于"**Binary Releases**"下面的"**Compatibility Matrix**"(兼容性矩阵)表格显示了**Hibernate**各组件之间的相互兼容关系)。

同样, 下载这个文件, 将其解压到存放**Hibernate**的相关目录中。

如果你下载链接时遇到麻烦, 可能是因为网站正在维护, 处于不稳定的状态, 所以就看不到你要的文件。如果真遇到这样的情况, 你可以点击"**Binary Releases**"方框下面的"**Browse all Hibernate downloads**"链接, 滚动页面, 查找需要下载的内容。**Hibernate**项目非常活跃, 所以发生这种情况比你想象的要更频繁。

建立项目层次结构

虽然起步只是一小步，但是，当我们开始设计数据结构，并创建用于表示它们的Java类和数据库表（**database table**），再配合所有配置文件和控制文件，将全部内容整合起来以发挥作用时，我们最后还是会得到一大堆文件。所以，起步时我们得先有个良好的组织体系。从前面下载的工具和相关的支持库之间就可以看出，有许多文件得好好加以组织。幸运的是，**Maven Ant Tasks**可以帮我们下载和管理所有的外部依赖文件。

为何在意

如果你通过扩展本书的示例而做出了一些很酷的东西，想将它们转换成实用的应用程序，那么你的代码从一开始就得有良好的组织。更重要的是，如果你按照这里所说的方式组织，我们在示例中给你的命令和指令才会有意义，才能起实际作用。书中有很多示例也彼此相关，因此一开始走对路是很重要的。

如果你想跳过这一段去看后面的示例，或者不想输入一些很长的示例程序代码和配置文件，则可以从本书网站下载各章示例的“最

终”版本。这些下载到的文件的组织方式的确如这里所述。我们强烈建议你下载示例代码，并将它们作为你阅读本书时的参考。

应该怎么做

以下介绍如何建立一个空的项目层次结构，如果你还没有下载“最终的”示例：

- 1.在硬盘目录中选择一个位置，作为演练这些示例的目录，然后创建一个新的目录，从现在开始，我们就称这个目录为你的项目目录。

- 2.进入该目录，创建名为src及data的子目录。Java源代码和相关资源的层次结构会放在src目录中。我们的构建过程在编译代码时，会将结果放在编译时创建的classes目录下，然后把运行时需要的任何资源都复制到这个目录中。data目录用于存放HSQLDB数据库相关的文件。

- 3.我们打算创建的示例类都将放在com.oreilly.hh（harnessing Hibernate）包中，而将Hibernate生成的数据bean都放在com.oreilly.hh.data包中，以便将它们与我们手工创建的类分离开来，所以我们在src目录下创建这些目录。在Linux和Mac OS X上，可以使用以下命令来创建目录：

```
mkdir -p src/com/oreilly/hh/data
```

在项目目录中执行这个命令，一步就可以完成目录的创建。

这时，你的项目目录结构应该如图1-5所示。

注意：与本书第1版相比，这一版的目录结构要简单得多，似乎不值得一提！

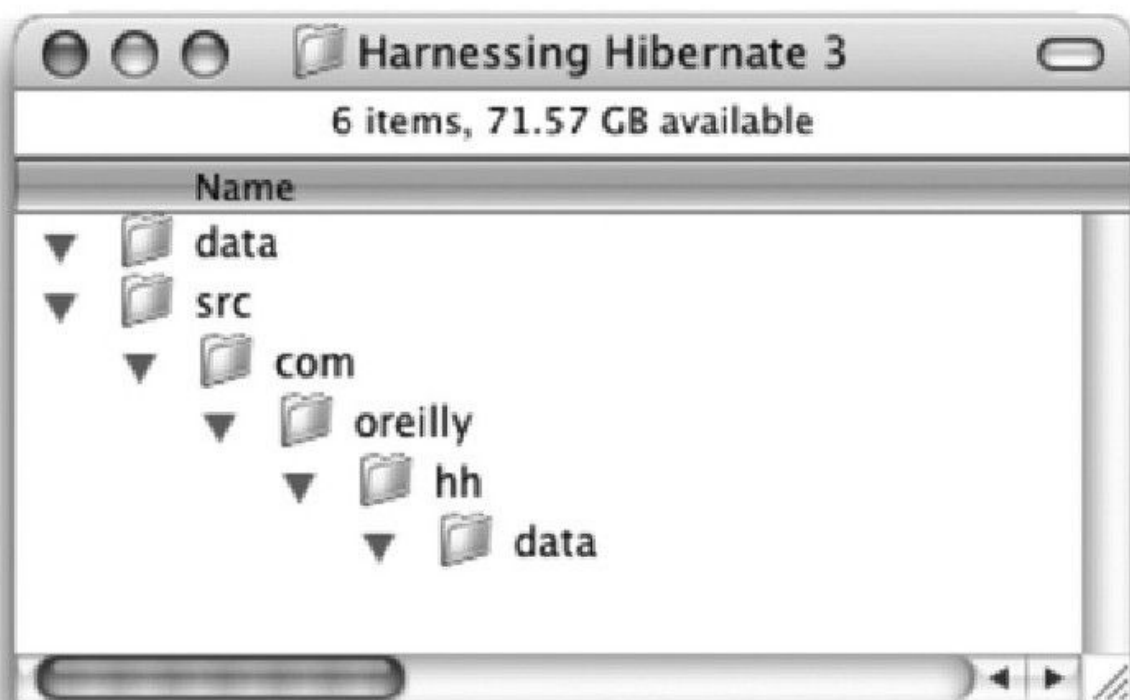


图 1-5 最初的项目目录结构

快速测试

在我们真正使用Hibernate来做些有用的工作以前，还应该检查一下其他支持库是否存在和可以使用。我们先从整个项目都会用到的Ant

配置文件开始，告诉Ant我们把要用的文件放在哪里，同时让Ant启动HSQLDB数据库图形界面。这可以证明Maven Ant Tasks能够找到并下载示例依赖的库，可以访问数据库图形界面，通过这个界面我们可以查看Hibernate为我们创建的真实数据。此时只是作为一个基本的完整性检查，以确认没有少什么东西，保证我们为继续学习做好准备。

打开你最喜欢的文本编辑器，在项目目录最顶层创建一个名为build.xml的文件。将例1-1的内容输入到这个文件中。

例1-1: Ant构建（build）文件

```
<?xml version="1.0"?>❶
<project name="Harnessing Hibernate 3 (Developer's Notebook
Second Edition) "
  default="db"basedir="."
  xmlns: artifact="antlib: org.apache.maven.artifact.ant">❷
  <!--Set up properties containing important project directories-->❸
  <property name="source.root"value="src"/>
  <property name="class.root"value="classes"/>
  <property name="data.dir"value="data"/>
  <artifact: dependencies pathId="dependency.classpath">❹
  <dependency
groupId="hsqldb"artifactId="hsqldb"version="1.8.0.7"/>
  <dependency groupId="org.hibernate"artifactId="hibernate"
version="3.2.5.ga">
  <exclusion groupId="javax.transaction"artifactId="jta"/>
  </dependency>
  <dependency groupId="org.hibernate"artifactId="hibernate-tools"
version="3.2.0.beta9a"/>
  <dependency groupId="org.apache.geronimo.specs"
artifactId="geronimo-jta_1.1_spec"version="1.1"/>
  <dependency groupId="log4j"artifactId="log4j"version="1.2.14"/>
  </artifact: dependencies>
  <!--Set up the class path for compilation and execution-->
  <path id="project.class.path">❺
  <!--Include our own classes, of course-->
```

```

    <pathelement location="${class.root}"/> ❸
    <!--Add the dependencies classpath-->
    <path refid="dependency.classpath"/> ❹
  </path>
  <target name="db" description="Runs HSQLDB database management UI
  against the database file--use when application is not running">
❷
    <java classname="org.hsqldb.util.DatabaseManager"
    fork="yes">
    <classpath refid="project.class.path"/>
    <arg value="-driver"/>
    <arg value="org.hsqldb.jdbcDriver"/>
    <arg value="-url"/>
    <arg value="jdbc: hsqldb: ${data.dir}/music"/>
    <arg value="-user"/>
    <arg value="sa"/>
    </java>
  </target>
</project>

```

输入时要小心那些标点符号，对于自结束的（self-closing）XML 标签更要小心（是“/>”，而不是“>”）。如果输入错误，代价就是运行Ant时得到解析错误的信息。如果你不想手工输入这些文件的内容，可以从本书网站下载这些文件。如果你正在阅读的是PDF电子文档，也可以剪切和粘贴代码，但需要去掉一些无关的排版字符。

如果你以前没有看过Ant构建文件，以下简单介绍可以帮助你熟悉它。（如果你想查看更多的细节，可以访问文档 <http://ant.apache.org/manual/index.html>。）

❶第1行只是声明这个文件的类型是XML。如果你以前在其他情况下用过XML，就应该很习惯这种文件格式；如果没有，那就再看一次

（Ant目前不需要这一行，但大多数XML解析器都需要，因此养成这种习惯是一件好事）。

❷Ant的构建文件总包含一个project定义。default属性用于告诉Ant，如果在命令行没有指定要构建任何目标（target），Ant要默认构建哪个目标（后面会定义）。basedir属性用于指定所有路径计算都对应于哪个目录。我们可以不指定这个属性，因为在默认情况下总是将build.xml所在的目录作为相对目录的基本目录，但是明确指定类似的基本设置是一种好习惯。在这个project元素中需要注意的一个重要事情是，xmlns: artifact是专门为Maven Ant Tasks提供的命名空间定义。该命名空间定义使用了artifact前缀，这样就可以在构建文件中使用Maven Ant Tasks了（后面有具体的使用方法）。

❸接下来的几行定义了3个属性，我们可以在构建文件的其他部分中，通过名称来引用相应的属性。基本上，我们在此是为项目中各个部分用到的重要目录定义其符号名称。这样做并非必要（尤其是目录名称非常简单时），不过这也是一种好习惯。至少，如果你得修改这些目录中的某个目录时，只需要修改构建文件中的一个地方，而不用进行繁琐的搜索和替换操作。

❹artifact: dependencies元素来源于Maven Ant Tasks，你（以及Ant）可以通过artifact: 前缀判断出这一点。在这个元素中，我们定义了一套依赖，项目需要它们才可以编译和执行。这些依赖对应于Maven

2 Repository（位于<http://repol.maven.org/maven2>）中央仓库中的JAR文件（或其他生成文件）。每个artifact（工件）由一个groupId、artifactId以及version号进行惟一标识。在这个项目中，我们需要依靠Hibernate、HSQLDB、Log4J以及JTA API。当Maven Ant Tasks遇到这些依赖声明时，就会从Maven 2中央仓库中按照需要将每个artifact下载到你的本地Maven 2仓库中（在`~/.m2/repository`目录下）。如果你对这一区段的配置内容还没有什么感觉，大可不必担心，在稍后几页的内容中，我们将深入探索相关的细节。如果需要，你可以对这一区段的配置作出修改（只要修改version值），以便使用这些依赖的程序包的更新版本（因为在本书交付印刷之后，可能会有新版本推出）。不过你可以放心，本书印刷出版后，书中的例子将一定能够正常运行，因为不论你什么时候研究这些例子，Maven仓库可以确保我们测试过的版本总是可用的。我们之所以在本书中采用Maven Ant Tasks，这也是很大的一部分原因。

❶class-path区段的用途很明确。这个功能正是我每次做Java项目时，几乎总要为其配置一个简单的Ant构建文件的原因。无论做什么规模的项目，总是需要将很多第三方程序库放到类路径中，而且还得确保编译时和运行时的配置都完全一样。Ant使得这一工作变得很简单。我们定义了一个path，有点像一个属性，但是它知道应该如何解析、收集文件和目录信息。我们的类路径包含classes目录，经过编译的Java文件就放在这个目录中（这个目录现在还不存在；下一章会在构建处理

中多增加一个步骤以创建它），此外，也包含与`artifact: dependencies`元素中列出的依赖相对应的所有JAR文件。这正是编译和运行项目所需要的一切。

对于Java路径和类层次结构的理解和处理来说，**Ant**是一个很棒的工具，值得我们深入学习。

❹这一行的标点符号有点令人迷惑，但实际上可以划分成具有意义的几个部分。**Ant**可以使用置换（**substitution**）机制，将变量值插入到规则中。当你看到像“`${class.root}`”这样的语句时，这表示“寻找名为`class.root`的属性值，用这个值来置换这里的字符串”。所以，根据前面对`class.root`的定义，置换的结果就好像是在这里输入了：`<path element location="classes"/>`。那么，为什么要这么做呢？这是为了可以在整个构建文件中共享某个属性值，这样，如果需要修改相关的配置，则只需要关注一个地方就可以了。在大型的复杂项目中，这种组织和管理是很重要的。

❺我们在前面看到的`artifact: dependencies`元素使用它的`pathId`属性，将所有声明的依赖组装到一个名为`dependency.classpath`的路径中。这里，我们将`dependency.classpath`的内容附加到`project.class.path`中，以便在编译和运行时可以找到我们用**Maven**取回的依赖包。

⑧最后，这些前期工作做好以后，我们就能够定义第一个构建目标了。一个目标其实就是一系列任务，为了完成项目的目标，必须按顺序执行任务。典型的构建目标就是做些类似于编译代码、运行测试、为发布而打包文件等各种事情。可以从Ant内建的一组丰富的功能中选择任务，而像Hibernate这样的第三方工具则可以扩展Ant，以便提供它们自己的任务，这在下一章就会看到。我们的第一个构建目标是db，它会运行HSQLDB的图形界面，让我们查看示例数据库。我们可以用Ant内建的java任务来完成这项工作，它会为我们运行Java虚拟机，并配置好我们需要的启动类、参数（argument）以及属性。

就此例而言，我们想要调用的类是org.hsqldb.util.DatabaseManager，在HSQLDB JAR中可以找到这个类（Maven Ant Tasks将为我们管理这个依赖）。将fork属性设置为"yes"，这是告诉Ant使用另一个单独的虚拟机，而不是默认的虚拟机，因为默认虚拟机得花费比较长的时间，而且通常没有这个必要。在这个例子中，这一点很重要，因为我们想让数据库管理器GUI一直保持运行，直到我们关掉它，但是，如果在Ant自己的VM中运行，就达不到这样的效果。

如果你的数据库GUI弹出后，就马上消失，请再次检查你的java任务的fork属性。

你可以看到我们先是如何告诉java任务，有关之前已经配置好的类路径信息的（这是我们所有的构建目标都得使用的配置属性）。然后，我们为数据库管理器提供了很多参数，告诉它使用标准的HSQLDB JDBC驱动程序，到哪儿去找数据库，以及使用的用户名是什么。我们在data目录中指定了一个名为music的数据库。这个目录当前还是空的，所以HSQLDB会在我们第一次使用时创建这个数据库。新数据库默认的“系统管理员”用户名是sa，最初的配置是一开始不需要密码。显然，如果你计划让数据库在网络上也可以使用（HSQLDB能够做得到），就需要设置密码。我们不需要做这些花哨的事，所以现在先不用理会其他配置。

OK，让我们试一下吧！保存好这个文件，在你的顶级项目目录（build.xml所在的目录）的命令提示行中输入以下命令：

```
ant db
```

（或者，因为我们将db设置为默认的构建目标，你也可以只输入ant）在Ant开始运行以后，如果一切顺利，你会看到像下面这样的输出结果：

```
Buildfile: build.xml
Downloading: hsqldb/hsqldb/1.8.0.7/hsqldb-1.8.0.7.pom
Transferring 0K
Downloading: org/hibernate/hibernate/3.2.5.ga/hibernate-
3.2.5.ga.pom
Transferring 3K
Downloading: net/sf/ehcache/ehcache/1.2.3/ehcache-1.2.3.pom
```

Transferring 19K
Downloading: commons-logging/commons-logging/1.0.4/commons-logging-1.0.4.pom
Transferring 5K
Downloading: commons-collections/commons-collections/2.1/commons-collections-2.1.pom
Transferring 3K
Downloading: asm/asm-attrs/1.5.3/asm-attrs-1.5.3.pom
Transferring 0K
Downloading: dom4j/dom4j/1.6.1/dom4j-1.6.1.pom
Transferring 6K
Downloading: antlr/antlr/2.7.6/antlr-2.7.6.pom
Transferring 0K
Downloading: cglib/cglib/2.1_3/cglib-2.1_3.pom
Transferring 0K
Downloading: asm/asm/1.5.3/asm-1.5.3.pom
Transferring 0K
Downloading: commons-collections/commons-collections/2.1.1/commons-collections-2.1.1.pom
Transferring 0K
Downloading: org/hibernate/hibernate-tools/3.2.0.beta9a/hibernate-tools-3.2.0.beta9a.pom
Transferring 1K
Downloading: org/hibernate/hibernate/3.2.0.cr5/hibernate-3.2.0.cr5.pom
Transferring 3K
Downloading: freemarker/freemarker/2.3.4/freemarker-2.3.4.pom
Transferring 0K
Downloading: org/hibernate/jtidy/r8-20060801/jtidy-r8-20060801.pom
Transferring 0K
Downloading: org/apache/geronimo/specs/geronimo-jta_1.1_spec/1.1/geronimo-jta_1.1_spec-1.1.pom
Transferring 1K
Downloading: org/apache/geronimo/specs/specs/1.2/specs-1.2.pom
Transferring 2K
Downloading: org/apache/geronimo/genesis/config/project-config/1.1/project-config-1.1.pom
Transferring 14K
Downloading:
org/apache/geronimo/genesis/config/config/1.1/config-1.1.pom
Downloading:
org/apache/geronimo/genesis/config/config/1.1/config-1.1.pom

Downloading:
org/apache/geronimo/genesis/config/config/1.1/config-1.1.pom
Transferring 0K
Downloading: org/apache/geronimo/genesis/genesis/1.1/genesis-1.1.pom
Downloading: org/apache/geronimo/genesis/genesis/1.1/genesis-1.1.pom
Downloading: org/apache/geronimo/genesis/genesis/1.1/genesis-1.1.pom
Transferring 6K
Downloading: org/apache/apache/3/apache-3.pom
Downloading: org/apache/apache/3/apache-3.pom
Downloading: org/apache/apache/3/apache-3.pom
Transferring 3K
Downloading: log4j/log4j/1.2.14/log4j-1.2.14.pom
Transferring 2K
Downloading: org/hibernate/hibernate-tools/3.2.0.beta9a/hibernate-tools-3.2.0.beta9a.jar
Transferring 352K
Downloading: org/hibernate/jtidy/r8-20060801/jtidy-r8-20060801.jar
Transferring 243K
Downloading: commons-collections/commons-collections/2.1.1/commons-collections-2.1.1.jar
Transferring 171K
Downloading: commons-logging/commons-logging/1.0.4/commons-logging-1.0.4.jar
Transferring 37K
Downloading: antlr/antlr/2.7.6/antlr-2.7.6.jar
Transferring 433K
Downloading: org/apache/geronimo/specs/geronimo-jta_1.1_spec/1.1/geronimo-jta_1.1_spec-1.1.jar
Transferring 15K
Downloading: net/sf/ehcache/ehcache/1.2.3/ehcache-1.2.3.jar
Transferring 203K
Downloading: asm/asm/1.5.3/asm-1.5.3.jar
Transferring 25K
Downloading: freemarker/freemarker/2.3.4/freemarker-2.3.4.jar
Transferring 770K
Downloading: dom4j/dom4j/1.6.1/dom4j-1.6.1.jar
Transferring 306K
Downloading: asm/asm-attrs/1.5.3/asm-attrs-1.5.3.jar
Transferring 16K
Downloading: cglib/cglib/2.1_3/cglib-2.1_3.jar
Transferring 275K

```
Downloading: hsqldb/hsqldb/1.8.0.7/hsqldb-1.8.0.7.jar
Transferring 628K
Downloading: log4j/log4j/1.2.14/log4j-1.2.14.jar
Transferring 358K
Downloading: org/hibernate/hibernate/3.2.5.ga/hibernate-
3.2.5.ga.jar
Transferring 2202K
db:
```

一大堆下载文件信息表明**Maven Ant Tasks**完成了它的任务，为我们下载了指定的文件（包括**HSQldb**和**Hibernate**）以及所有依赖的库文件。这一过程可能需要花费一段时间才能完成（取决于网络连接的速度和服务器的情况），但是它只进行一次。下一次再运行**Ant**时，**Maven Ant Tasks**就会注意到你的本地库中已经包含了所有这些文件，就会默认继续执行其他需要完成的任务了。

在所有下载完成以后，**Ant**会打印输出"**db:**"，表明它正开始执行你请求的目标。一会儿，你应该会看到**HSQldb**的图形界面，如图1-6所示。我们的数据库现在还没有什么东西，所以，除了命令能运行以外，没有其他内容。窗口左上部的树形视图是数据库中可以浏览的各种数据表和字段。就目前而言，只要确认最上层是否为"**jdbc:hsqldb:data/music**"就可以了。



图 1-6 HSQLDB数据库管理器界面

如果你喜欢，可以浏览一下菜单，但不要对数据库进行任何修改。在完成以后，选择"File" → "Exit"，窗口就会关闭，同时Ant将报告以下信息：

```
BUILD SUCCESSFUL
Total time: 56 seconds
```

"Total time"是你运行这个数据库管理器程序所用的时间，所以其值是变化的（刚才我们在Ant的构建文件中为java任务设置了fork属性，所以直到关闭数据库以前，Ant会一直等待）。此时，如果你查看data目录，你会发现HSQLDB已经创建了一些文件来保存数据库信息：

```
%ls data
music.log music.properties music.script
```

你甚至可以查看这些文件的内容。和大多数数据库系统不同，**HSQLDB**是以人类可读的格式来存储数据的。`.properties`文件中包含一些基本设置，而`.script`文件中的数据则以SQL语句的形式保存。`.log`文件用于当应用程序崩溃或没有正常关闭数据库而强行退出时，可以重新构建一致的数据库状态。现在，你在这些文件中看到的都是基本定义，都是默认值；以后我们开始创建数据表并添加数据时，就可以看到这些文件会有所变化。这也可以作为一种有用的调试手段来进行基本的完整性检查，甚至比启动图形界面执行查询还要快。

注意：能够直接读取**HSQLDB**数据库文件的内容，是一件诡异但有趣的事。

发生了什么事

目前我们已经成功运行了第一个示例，建立了我们的项目构建文件，现在解释**Ant**如何取回这个项目必需的依赖文件。我们再查看一下示例的`build.xml`文件中的`artifact: dependencies`元素（如例1-2所示）。

例1-2: `artifact: dependencies`元素

```
<artifact: dependencies pathId="dependency.classpath">
  <dependency
groupId="hsqldb"artifactId="hsqldb"version="1.8.0.7"/>
  <dependency groupId="org.hibernate"artifactId="hibernate"
version="3.2.5.ga">
    <exclusion groupId="javax.transaction"artifactId="jta"/>
  </dependency>
```

```
<dependency groupId="org.hibernate"artifactId="hibernate-tools"
version="3.2.0.beta9a"/>
<dependency groupId="org.apache.geronimo.specs"
artifactId="geronimo-jta_1.1_spec"version="1.1"/>
<dependency groupId="log4j"artifactId="log4j"version="1.2.14"/>
</artifact: dependencies>
```

如果你以前从没有用过Maven，这个例子看起来非常令人费解。我们先定义一些术语。首先是artifact。一个artifact就是项目生成的一个文件，它可以是任意类型的文件—Web应用程序的WAR文件、企业应用程序的EAR文件或是简单的JAR文件。对于我们的用途来说，我们使用的是JAR artifact，如果你不在dependency元素中指定它，其默认的类型就是jar，非常方便。

artifact有4个属性：groupId、artifactId、version以及type。例如，我们需要org.hibernate组内的hibernate artifact，其版本是3.2.5.ga，文件类型是jar。Maven将使用这些标识符在位于<http://repo1.maven.org/maven2/>的中央Maven 2库中来查找定位适当的依赖文件。使用这些值，Maven将通过以下模式来尝试定位Hibernate的JAR：<repositoryUrl>/<groupId>/<artifactId>/<version>/<artifactId>-<version>.<type>，其中，groupId中的点号“.”将转换成URL的路径分隔符。使用这样的模式，就可以定位Hibernate和HSQLDB依赖的JAR文件了，如例1-3所示。

例1-3：项目依赖文件的URL

```
http://repo1.maven.org/org/hibernate/hibernate/3.2.5.ga/hibernate-3.2.5.ga.jar  
http://repo1.maven.org/hsqldb/hsqldb/1.8.0.7/hsqldb-1.8.0.7.jar
```

在build.xml文件中，我们没有在Hibernate依赖声明中包含JTA依赖。这是因为Hibernate库使用的是一个非自由（nonfree）的JAR文件，所以Maven 2公共库中没有这个文件。这个项目没有使用Sun提供的标准JTA API JAR，而是使用了Apache Geronimo项目创建的JTA API。geronimo-jta_1.1_spec是Java Transaction API的一个自由的开源实现版本。这个替换是通过在Hibernate 3.2.5.ga的依赖声明中使用exclusion元素来实现的，并随后明确声明了需要使用Geronimo JTA spec 1.1。

好了，我们已经介绍了Maven Ant Tasks如何从Maven库中取回依赖文件，但是，如果已经下载好了这些依赖文件，又会怎么样呢？Maven Ant Tasks将所有的依赖文件都下载到一个本地的Maven库中，由Maven负责维护这个本地库，使其结构与远程Maven库的结构保持一致。当需要检查某个artifact时，Maven先检查本地库，如果本地库没有这个artifact时，再从远程库中请求该artifact。这意味着，如果同时有20个项目都引用了同一版本的Hibernate，从Maven远程库中只下载一次Hibernate的依赖artifact，所有20个项目都会引用保存在本地Maven库中的同一个副本。那么，这个不可思议的本地Maven库是放在哪里呢？最简单的回答就是，在build.xml文件中添加一个目标。在原来的Ant build.xml的末尾添加以下目标，如例1-4所示。

例1-4: 打印输出依赖类的路径

```
<target name="print-classpath" description="Show the dependency
class path">
  <property name="class.path" refid="dependency.classpath"/>
  <echo>${class.path}</echo>
</target>
```

运行这个目标，将生成以下输出结果：

```
%ant print-classpath
Buildfile: build.xml
print-classpath:
[echo]~\.m2\repository\commons-logging\commons-logging\1.0.4\
commons-logging-1.0.4.jar; \
~\.m2\repository\dom4j\dom4j\1.6.1\dom4j-1.6.1.jar; \
~\.m2\repository\cglib\cglib\2.1.3\cglib-2.1.3.jar; \.....
```

运行该目标，输出结果会随你正在使用的操作系统的不同而有所变化，不过，你可以看到，依赖类的路径都引用了本地Maven库。在运行Windows XP的计算机上，这一路径可能位于C: \Documents and Settings\Username\.m2\repository；在运行Windows Vista的计算机上，这一路径则可能位于C: \Users\Username\.m2\repository；而在Unix和Macintosh上，该路径将会是~/.m2/repository目录。

你也可能会注意到，在类路径中列出的依赖文件的数目要比我们在build.xml的artifact: dependency元素中声明的多。这些额外的依赖就是所谓的可传递的依赖（transitive dependency），它们是你明确声明的依赖文件所需要的依赖。例如，Hibernate需要依赖CGLib、EHCache、

Commons Collections以及其他程序库。对Maven的详细介绍超出了本书讨论的范围，我在这里只对Maven Ant Tasks如何为你的项目构造整套依赖结构提供一些提示。如果在构建好一个示例后，你查看一下本地Maven库，你会注意到在每个JAR artifact旁边会有一个扩展名为.pom的项目对象模型（POM）文件。POM是Maven构建系统和仓库的基础，每一个POM描述了一个artifact和它的依赖。Maven Ant Tasks使用这种元数据（metadata）来构造一个描述依赖传递关系的树状结构。换句话说，Maven Ant Tasks并不只是负责下载Hibernate，它们还负责下载Hibernate依赖的所有文件。

你实际需要知道的是它的工作原理就是这样的。

接下来做什么

感谢Maven Ant Tasks，有了它，就不用像本书的最初版本那样亲自查找、下载、解压软件，并组织好它们。现在你可以从一个更高的起点来开始使用Hibernate，在下一章中你可以看到，我们的进展会非常快速。你将看到Hibernate已经为你写好了Java代码！数据库模式

（database schema）也好像是从天而降一样（或者，至少是来自于产生Java程序代码的同一个XML映射表）！真正的数据表和数据也会出现在HSQLDB管理器界面中（或者，至少是示范数据）！

听起来令人兴奋吧？嗯，和你以前的开发方法相比如何？我们继续研究下去，把Hibernate的强大功能唤醒吧。

为何无法工作

如果你没有看到数据库管理器窗口，而是出现了错误信息，那么应该尝试弄清楚是否是文件构建出了问题；是否是安装设置Ant或项目层次结构出错；是否是因为访问Internet困难，而不能从Maven库下载到依赖文件；或是其他什么原因。再次确认所有东西都安排妥当，并且按照前面介绍的方式正确地安装。如果是自己输入的文件有问题，可以考虑下载示例程序。

第2章 映射简介

我们已经开始使用**Hibernate**了。不过现在，我们要先停下向前迈进的脚步，站在一定高度观察一下**Hibernate**的全貌，以免迷失在**Hibernate**安装和配置的繁文缛节之中，这是非常必要的。像Java这样的面向对象语言提供了强大而方便的抽象层，可以在运行时以对象（类的实例）的形式来处理信息。这些对象之间可以通过各种方式链接起来，除了它们本身所拥有的原始数据外，还可以包含规则和行为。但是，当程序运行结束时，所有对象都会消失得无影无踪。

对于那些在程序多次运行之间需要保存下来的信息，或者是需要在不同程序或系统之间共享的信息，实践证明关系数据库是难以击败的最佳解决方案。关系数据库具有高度的可伸缩性、可靠性、高效性以及灵活性。所以，我们需要有一种方式可以把信息从SQL数据库中提取出来，再将其转换成Java对象，反之亦然。

实现这一功能有许多不同的方法，从完全手工的数据库设计和编码，到高度自动化的工具，都有相应的解决方案。这个普遍性问题就是人们所谓的对象/关系数据库映射（Object/Relational Mapping）问题，而**Hibernate**正是Java中一种轻量级的O/R映射服务。

所谓“轻量级”（**lightweight**）是指，和其他一些可用的工具相比，**Hibernate**的设计相当易于学习和使用，同时对系统资源的需求也在合理的范围内。虽然如此，**Hibernate**还是设法达到了用途广泛而又有技术的深度。**Hibernate**的设计者对实际项目中需要完成的工作进行了细致的研究，并对这些工作提供良好的支持。

Hibernate的使用方法有很多种，这要取决于你开始时着手的对象。如果需要交互的数据库已经存在，则可以用**Hibernate**的一些工具对现有的数据库模式（**schema**）进行分析，以此作为映射（**mapping**）的起点，之后，它再帮助你编写一些用于表示那些数据的**Java**类。如果你已经有了**Java**类，并希望把这些类的实例中的数据保存在数据库中，则可以从这些**Java**类开始，用**Hibernate**工具生成相应的映射文件，再生成用于创建数据库表的模式脚本。

在本书中，我们将要带领你从一个全新的项目开始，没有现成的**Java**类或数据库表，让**Hibernate**帮助你生成这些类和数据库表。当像这样从头做起时，最佳的切入点就是二者之间的一个中间点，也就是我们打算在程序对象和存储这些对象的数据表之间使用的映射的抽象定义。附录E就如何深入学习**Hibernate**列举了一些建议；如果你愿意使用**Eclipse**，第11章还得介绍如何在**Eclipse**中使用**Hibernate**。

对于那些已经习惯处理**Java**对象，而对抽象模式比较陌生的开发人员来说，他们在熟悉这种方法之前，可能会遇到些小麻烦，或许只

是为一个外部的XML文件费神而已。第7章会演示如何使用Java 5的标注，在你的数据模型类中嵌入映射信息。搞清楚基于XML的映射是很重要的，所以我们就从它开始学习Hibernate。

在我们的示例中，我们将处理一个数据库，用它来驱动一个应用程序界面，用户可以保存许多个人音乐收藏数据，还可以方便地进行搜索、浏览和欣赏音乐（第1章最后创建了一些数据库文件，从其中的文件名你或许可以猜到这些吧）。

编写映射文档

传统的Hibernate使用XML文档来维护Java类和关系数据库表之间的映射关系。这种映射文档设计为可由开发人员阅读和手工编辑的。你也可以用图形化的CASE（Computer Aided Software Engineering，计算机辅助软件工程）工具（例如Togethor（[\[1\]](#)）、Rose（[\[2\]](#)）以及Poseidon（[\[3\]](#)））来建立表示数据模型的UML图表，再将这些图表输入到AndroMDA（[\[4\]](#)）（<http://www.andromda.org/>）中，就可以生成相应的Hibernate映射文档。前面提到的Hibernate Tools包也可以为你提供这些功能（第11章将会演示在Eclipse中使用它们是多么容易）。

要牢记，Hibernate及其扩展工具可以让你用其他方式进行开发。如果你已经有了Java类或数据，也可以从它们开始着手。我们还会学

习一种更新的方法—**Hibernate**标注，它可以让你完全脱离**XML**映射文档，这种方法将在第7章加以介绍。

这里，我们先手工编写**XML**映射文档，这是一种相当实用的方法。我们要为曲目（**Track**）对象创建映射文件。曲目就是各个可以单独欣赏的乐曲，或是可以收录在乐曲集或演奏列表中的乐曲。首先，我们要记录曲目的标题、存储实际音乐的文件路径、它的播放时间、曲目添加进数据库的日期以及播放曲目应该用的音量（不同乐曲在录制时采用的音量不见得相同，总用预设的默认音量播放不一定合适）。

为什么选择这个示例

你可能根本不需要创建一个新的系统来保存音乐曲目，但在建立这个示例的映射文档时所涉及的概念和处理过程，可以在你的实际项目中加以应用。

应该怎么做

注意：你可能会觉得这一映射文件中包含了太多的信息。正如你所看到的，确实如此，可以用它来创建很多有用的项目资源。

打开你喜欢用的文本编辑器，在前面1.7节中创建的src/com/oreilly/hh/data目录下创建一个名为Track.hbm.xml的文件（如果你跳过了这一节，就得回去再阅读一下1.7节，因为本示例要依赖当时所建立的项目目录结构和工具）。在该映射文档中输入例2-1所示的内容。

例2-1：曲目对象的映射文档：Track.hbm.xml

```
<?xml version="1.0"?>
<! DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">❶
<hibernate-mapping>
<class name="com.oreilly.hh.data.Track"table="TRACK">❷
<meta attribute="class-description">❸
Represents a single playable track in the music database.
@author Jim Elliott (with help from Hibernate)
</meta>
<id name="id"type="int"column="TRACK_ID">❹
<meta attribute="scope-set">protected</meta>
<generator class="native"/>❺
</id>
<property name="title"type="string"not-null="true"/>❻
<property name="filePath"type="string"not-null="true"/>
<property name="playTime"type="time">❼
<meta attribute="field-description">Playing time</meta>
</property>
<property name="added"type="date">
<meta attribute="field-description">When the track was created
</meta>
</property>
<property name="volume"type="short"not-null="true">
<meta attribute="field-description">How loud to play the track
</meta>
</property>
</class>
</hibernate-mapping>
```

❶最前面的3行是有效的XML文档所必需的预定义部分，用于声明该XML文档符合Hibernate映射使用的文档类型定义。实际的映射内容将位于hibernate-mapping标签内部。

❷我们正在为一个类（`com.oreilly.hh.data.Track`）定义它的映射，这个类的名称和包名与已经创建的文件名称和路径是一致的。但这种关系不是必需的，可以在一个映射文档中为任意数量的类定义它们的映射，为映射文件起任意名字，并把它放在任何地方，只要告诉Hibernate怎么找到这个文件就可以了。不过，根据映射到的类来命名映射文件，并将它和映射到的类放在一起，这一惯例带来的好处是：当需要使用这个类时，Hibernate能够自动定位该映射文档。当类的数量不多时，这样可以简化Hibernate的配置和使用。

如果映射类的数量比较多，那么你可能会希望使用XML格式的Hibernate配置文件，再从这个配置文件中引用所有的映射文档，而不必像本书第1版那样，在所有示例源代码中再涉及它们（从下一章开始，就会看到我们换用了这种新方法）。此外，在使用基于XML的配置文件时，将各个映射文档和Hibernate配置文件放在一起，而不是和映射类分散地保存，这样做更有意义。

在class元素的开始标签中，我们指定这个类要保存在一个名为TRACK的数据库表中。

❸ **meta** 标签并不会直接影响映射，相反，这个标签为使用其他不同工具而提供额外的信息。在这个例子中，通过将 **attribute** 属性值指定为 "class-description"，我们告诉 Java 代码生成工具：需要为 **Track** 类关联什么样的 **JavaDoc** 文本信息。这个标签完全是可选的，在本章稍后的“生成 Java 类”一节中，就会看到包含 **JavaDoc** 信息的结果。

❹ 映射内容的其他部分用于设定我们想保存到数据库中的信息，也就是类的属性以及数据库表中与之相关的字段。虽然我们在介绍这个例子时没有提及，但每个曲目对象都需要一个 **id** 属性。按照数据库的最佳实践标准，我们应该用一个没有意义的代理键（**surrogate key**，一个没有语义含义的值，只用来标识特定的数据行）。在 **Hibernate** 中，**key/id**（键/属性）之间的映射关系是用 **id** 标签来设置的。我们准备使用一个 **int** 类型的值把 **id** 保存在数据库字段 **Track_id** 中。这里又包含了一个 **meta** 标签，用于和 Java 代码生成器进行通信，告诉它这个 **id** 属性的 **set** 方法应该具有 **protected** 类型的访问权限，因为应用程序代码本身没有必要去修改曲目对象的 **ID**。

❺ **generator** 标签用于设置 **Hibernate** 如何为新的实例创建其 **id** 值（注意，该标签与普通的对象/关系映射操作有关，而与 Java 代码生成器无关，而且也不经常使用代码生成器；**generator** 要比可选的 **meta** 标签发挥更多的功能）。在这个标签中可以从中选择很多不同的 **ID** 生成策略，甚至可以编写自己的策略。在这个例子中，我们告诉 **Hibernate** 使

用底层数据库最自然的主键生成方法（稍后我们会看到Hibernate如何知道我们正在使用什么数据库）。在HSQLDB中，会采用一个标识（identity）字段。

⑥在ID之后，我们只列举出关心的各种曲目属性。title是一个字符串属性，而且不能为null。filePath也有同样的属性设置，而除了volume以外，其他属性都可以为null。

⑦playTime是time类型，added是date类型，而volume是short类型。最后三个属性用了另一个新的meta属性"field-description"，用它为单个属性指定JavaDoc文本信息，但目前的代码生成器对此还有些限制。

该映射文件严格而简练地指定了我们需要表达的音乐曲目的各种数据，而且Hibernate也可以处理这种映射格式。接下来我们就看看Hibernate实际上是如何处理的（Hibernate能够表示更为复杂的信息，包括表示对象之间的关系，我们将在接下来的几章中加以介绍。附录E也讨论了一些与深入学习本书内容相关的方法）。

[1] <http://www.borland.com/us/products/together/index.html>.

[2] <http://www-306.ibm.com/software/awdtools/developer/thechnical/>.

[3] <http://genteware.com/index.php>.

[4] <http://www.andromda.org/>.

生成Java类

我们的映射文件中包含的是有关数据库、Java类以及它们之间的映射关系的信息。可以用它来帮助我们创建数据库表和Java类。首先，我们来看看Java类。

应该怎么做

你在第1章安装的Hibernate Tools中包含一个工具，可以生成匹配映射文档规定的Java源代码。只要用一个Ant任务就能方便地在Ant的构建文件中调用这一工具。编辑build.xml，把例2-2中的黑体部分添加进去。

例2-2: Ant构建文件（已经为生成Java代码而做了相应的更新）

```
<?xml version="1.0"?>
<project name="Harnessing Hibernate 3 (Developer's Notebook
Second Edition) "
  default="db"basedir="."
  xmlns: artifact="antlib: org.apache.maven.artifact.ant">
  <!--Set up properties containing important project directories-->
  <property name="source.root"value="src"/>
  <property name="class.root"value="classes"/>
  <property name="data.dir"value="data"/>
  <artifact: dependencies pathId="dependency.class.path">
  <dependency
groupId="hsqldb"artifactId="hsqldb"version="1.8.0.7"/>
  <dependency groupId="org.hibernate"artifactId="hibernate"
version="3.2.5.ga">
```

```

    <exclusion groupId="javax.transaction"artifactId="jta"/>
  </dependency>
  <dependency groupId="org.hibernate"artifactId="hibernate-tools"
version="3.2.0.beta9a"/>
  <dependency groupId="org.apache.geronimo.specs"
artifactId="geronimo-jta_1.1_spec"version="1.1"/>
  <dependency groupId="log4j"artifactId="log4j"version="1.2.14"/>
</artifact: dependencies>
<!--Set up the class path for compilation and execution-->
<path id="project.class.path">
  <!--Include our own classes, of course-->
  <pathelement location="${class.root}"/>
  <!--Add the dependencies classpath-->
  <path refid="dependency.class.path"/>
</path>
<!--Teach Ant how to use the Hibernate Tools-->
<taskdef name="hibernatetool"❶
classname="org.hibernate.tool.ant.HibernateToolTask"
classpathref="project.class.path"/>
<target name="db"description="Runs HSQLDB database management
UI
  against the database file--use when application is not
running">
  <java classname="org.hsqldb.util.DatabaseManager"
fork="yes">
    <classpath refid="project.class.path"/>
    <arg value="-driver"/>
    <arg value="org.hsqldb.jdbcDriver"/>
    <arg value="-url"/>
    <arg value="jdbc: hsqldb: ${data.dir}/music"/>
    <arg value="-user"/>
    <arg value="sa"/>
  </java>
</target>
  <!--Generate the java code for all mapping files in our source
tree-->
  <target name="codegen"❷
description="Generate Java source from the O/R mapping files">
    <hibernatetool destdir="${source.root}">
      <configuration>
        <fileset dir="${source.root}">
          <include name="**/*.hbm.xml"/>
        </fileset>
      </configuration>
    </hibernatetool>
  </target>
</project>

```

我们在构建文件中新添加了一个`taskdef`（任务定义）元素，以及一个用于创建文件的`target`元素：

❶这一行的任务定义教给Ant一个新技巧：它告诉Ant如何使用Hibernate Tools中包含的`hibernate tool`任务（有个专门的类为该目的提供帮助）。注意，它也指定了调用这个工具时所需的类路径，并引用`project.class.path`定义（这个定义包含了Maven Ant Tasks为我们管理的所有依赖文件）。当Ant需要使用`hibernate`工具时，通过这种办法，就可以找到它们。

❷`codegen`构建目标使用Hibernate Tools（`hbm2java`）模式来运行Hibernate的代码生成器，处理src源代码目录中找到的任何映射文件，生成相应的Java源代码。“`/**/*.hbm.xml`”这样的匹配模式（`pattern`）是说“在指定目录或其任何子目录下（无论有多深），任何以`.hbm.xml`结尾的文件”。

让我们先试一下吧！从你的项目目录的命令行中，输入以下命令：

```
ant codegen
```

你应该看到类似以下内容的输出（假设你运行过前面章节中`ant db`示例，运行那个例子会下载所有必需的依赖文件；如果你还没有运行

过那个例子，则此时在下载这些依赖文件时，会多显示许多行信息）：

```
Buildfile: build.xml
codegen:
[hibernatetool]Executing Hibernate Tool with a Standard
Configuration
[hibernatetool]1.task: hbm2java (Generates a set of .java files)
[hibernatetool]log4j: WARN No appenders could be found for
logger
(org.hibernate.cfg.Environment) .
[hibernatetool]log4j: WARN Please initialize the log4j system
properly.
BUILD SUCCESSFUL
Total time: 2 seconds
```

以上提示信息大致说的是，我们在第1章配置构建文件要安装log4j，但还没有建立它需要的配置文件，我们将在2.3节介绍解决这一问题的办法。现在，如果你去看看src/com/orielly/hh/data目录，就会发现出现了一个名为Track.java的新文件，其内容如例2-3所示。

例2-3：根据Track映射文档生成的Java源代码

```
package com.oreilly.hh.data;
//Generated Sep 2, 2007 10: 27: 53 PM by Hibernate Tools 3.2.0.b9
import java.util.Date;
/**
 *Represents a single playable track in the music database.❶
 *@author Jim Elliott (with help from Hibernate)
 */
public class Track implements java.io.Serializable{
❷
    private int id;
    private String title;
    private String filePath;
    /**
```

```

    *Playing time
    */
    private Date playTime;
    /**
    *When the track was created
    */
    private Date added;
    /**
    *How loud to play the track
    */
    private short volume;
    ❸
    public Track () {
    }
    public Track (String title, String filePath, short volume) {
        this.title=title;
        this.filePath=filePath;
        this.volume=volume;
    }
    public Track (String title, String filePath, Date playTime, Date
added,
        short volume) {
        this.title=title;
        this.filePath=filePath;
        this.playTime=playTime;
        this.added=added;
        this.volume=volume;
    }
    public int getId () {
        return this.id;
    }
    protected void setId (int id) {❹
        this.id=id;
    }
    public String getTitle () {
        return this.title;
    }
    public void setTitle (String title) {
        this.title=title;
    }
    public String getFilePath () {
        return this.filePath;
    }
    public void setFilePath (String filePath) {
        this.filePath=filePath;
    }
    /**
    **Playing time

```

```
*/
public Date getPlayTime () {
return this.playTime;
}
public void setPlayTime (Date playTime) {
this.playTime=playTime;
}
/**
**When the track was created
*/
public Date getAdded () {
return this.added;
}
public void setAdded (Date added) {
this.added=added;
}
/**
**How loud to play the track
*/
public short getVolume () {
return this.volume;
}
public void setVolume (short volume) {
this.volume=volume;
}
}
```

这个文件是怎么生成的？**Ant**会找出源代码树中所有以**.hbm.xml**结尾的文件（目前只有一个），把它们传给**Hibernate**的代码生成器，该代码生成器会分析映射文件，并生成一个满足**Track**映射文件中指定映射要求的**Java**类文件。很明显，这个工具可以为我们节省很多时间，并帮我们完成一些重复性的工作！

注意：**Hibernate**可以为我们节省许多时间并完成很多繁琐的操作。要不然，我们一定会被它们搞得晕头转向。

比较生成的Java源代码和映射文件中的映射规定（例2-1），你会有所收获。源代码以相应的包（`package`）声明作为开始，对于hbm2java而言，根据映射文件中指定的完全限定的类名，就可以容易地确定正确的包名：

❶ 类级的JavaDoc看起来应该很眼熟，因为它来自映射文档中meta标签的"class-description"。

❷ 字段声明是来自于映射文档中定义的id和property标签。源代码中所用到的Java类型都是源自于映射文档中的property类型声明。学习有关Hibernate支持的全部值类型，可以参阅附录E提到的资源。目前而言，映射文件内的类型和已生成的代码内的Java类型两者间的关系应该相当明确。

❸ 字段声明之后是三个构造函数的定义。第一个构造函数是创建实例时不用任何参数（如果想让你的类能够作为标准的bean来使用，例如在JSP（Java Server Page）内使用，这是这种数据类非常常见的用法）；第二个构造函数只需要提供映射文档中标明不得为null的值；最后一个构造函数是为所有属性赋值。注意，这些构造函数都没设置id属性的值，当我们从数据库把对象取出或者第一次将对象数据插入数据库时，Hibernate负责帮我们做这件事。

④同样，`setId()` 方法的访问类型是`protected`，与`id`的映射配置的要求是一样的。后面的`getter`和`setter`方法就没什么特别之处了，都是些照本宣科式的程序代码（我们都写过无数次了），这就是让Hibernate工具为我们生成这些代码的妙处所在。

如果你想使用Hibernate生成的代码作为起点，并在生成的Java类中添加某些业务逻辑或其他功能，一定要记住，你所做出的修改在下次运行代码生成器时，都会被“悄悄”地丢弃。在这样的项目中，你需要确保手工修改过的类不会被任何Ant的构建目标任务重新生成。一种常用的技巧是，把需要手工修改的类扩展成Hibernate生成的类。这也是为什么我们要将映射生成的Java类隔离到它们自己的代码包和子目录中的原因之一。

虽然此例中Hibernate为我们生成了数据类，但需要指出的一个要点是，Hibernate创建的`getter`和`setter`方法不仅仅是为了样子好看。对于你需要持久化的任何属性，在持久类中都必须具有`getter`和`setter`方法。这是因为Hibernate底层的持久化架构是通过反射机制来访问JavaBeans风格的属性的。如果你不希望类的属性是公共（`public`）的，它们可以不必是`public`的（即使属性被声明为`protected`或`private`，Hibernate还是有办法取得这些属性的值），但它们必须具有访问器和修改器方法，即`getter`（访问器）和`setter`（修改器）方法。这一点也是优秀的面向对象

设计应该遵循的准则；Hibernate团队希望把实际的实例变量（instance variable）的实现细节与底层持久化保存机制完全分离开来。

编制数据库Schema

刚才真简单，对吧？根据映射来创建数据库表也差不多是这样，你会很愉快地学习下去。和代码生成一样，只要利用映射文件，几乎所有工作都做完了。剩下的就是设置和运行schema生成工具。

应该怎么做

第一步是我们在第1章间接提到的一些事情。我们必须告诉Hibernate，我们要使用什么数据库，这样，Hibernate才知道应该使用什么SQL“方言”（dialect）。没错，SQL是标准，但每种数据库系统在标准SQL基础上，还都有各自在某些方面的扩展，有一套会影响到实际应用的特定功能和限制。为了解决这一现实问题，Hibernate提供了一组类来封装常见数据库环境的独特功能，这些类位于org.hibernate.dialect包中。你只需要告诉Hibernate想要使用哪一种数据库（如果你要使用的数据库是Hibernate目前尚未支持的，也可以自行实现必需的SQL方言）。

在我们的这个例子中，使用的是HSQLDB数据库，所以要用HSQLDialect。配置Hibernate最简单的方法就是创建一个名为

hibernate.properties的属性配置文件，再将它放在项目类路径的根目录中。在src目录的顶层创建这个文件，将例2-4的内容放进去。

例2-4：设置hibernate.properties的内容

```
hibernate.dialect=org.hibernate.dialect.HSQLDialect
hibernate.connection.driver_class=org.hsqldb.jdbcDriver
hibernate.connection.url=jdbc: hsqldb: data/music
hibernate.connection.username=sa
hibernate.connection.password=
hibernate.connection.shutdown=true
```

除了设置要使用的SQL方言以外，这个属性配置文件也会告诉Hibernate如何使用封装在HSQLDB数据库JAR文件内的JDBC驱动程序来建立数据库的连接，而且数据应该放在data目录下名为music的数据库中。在经过第1章的实践以后，对用户名和空密码（事实上，所有都是这个值）应该都很熟悉了。最后，我们告诉Hibernate，当处理完成后，应该显式关闭数据库连接，这是在嵌入模式下处理HSQLDB的一个artifact。如果不关闭连接的话，当工具退出时，对数据库的修改就不一定会写入到数据库中，所以你的数据库模式总是莫名其妙地为空。

如前所述，你也可以使用XML格式来保存配置信息，但就此处的简单需求而言，这样做没有什么价值。我们将在第3章介绍XML格式的配置方法。

你也可以把.properties文件存放在其他地方，并提供不同的文件名，或者以完全不同的方式把这些设置属性传递给Hibernate。但这里是Hibernate寻找配置文件的默认位置，因此这是个最方便的路径（或者，我猜想也是需要最少运行时配置的方法）。

我们也需要在Ant构建文件中加入一些新项，如例2-5所示，在build.xml文件末尾的</project>结束标签前增加了一些新的目标。

例2-5：生成schema所需要的Ant构建文件

```
<!--Create our runtime subdirectories and copy resources into
them-->
<target name="prepare" description="Sets up build structures">❶
  <mkdir dir="${class.root}"/>
  <!--Copy our property files and O/R mappings for use at
runtime-->
  <copy todir="${class.root}">
    <fileset dir="${source.root}">
      <include name="**/*.properties"/>
      <include name="**/*.xml"/>❷
    </fileset>
  </copy>
</target>
<!--Generate the schemas for all mapping files in our class
tree-->
<target name="schema" depends="prepare"❸
  description="Generate DB schema from the O/R mapping files">
  <hibernatetool destdir="${source.root}">
    <configuration>
      <fileset dir="${class.root}">
        <include name="**/*.hbm.xml"/>
      </fileset>
    </configuration>
    <hbm2ddl drop="yes"/>❹
  </hibernatetool>
</target>
```

❶首先，我们加了一个`prepare`构建目标，以供其他构建目标使用，而不是从命令行直接调用。其用途是在必要时创建`classes`目录，以便将编译好的Java代码放在这个目录中，再把`src`目录中找到的任何`.properties`文件和映射文件都复制到对应目录的`classes`层。这种层次式的复制是Ant相当棒的功能（使用特殊的“`**/*`”匹配模式），让我们可以定义和编辑源代码文件会使用到的相关资源，同时在运行时通过类加载器（`class loader`）来使用这些资源。

❷这一设置会让Ant复制所有找到的XML文件，而不仅仅是映射文档。虽然我们现在还不需要这样的配置文件，但以后当我们切换到基于XML的Hibernate配置文件时，这一设置就显得很重要了。

❸`schema`构建目标需要依赖`prepare`构建目标，让它把映射文档复制到正确的位置以供运行时使用。它会用`hbm2ddl`模式来调用Hibernate工具，让它们为在`classes`目录中找到的任何映射文档生成相应的数据库模式（如前所述，在下一章我们使用功能更强大的Hibernate XML配置文件时，这样就会更简单些）。

❹可以为`schema`生成工具指定很多参数，以配置其工作方式。在这个例子中，我们告诉`schema`生成工具在根据映射文档生成新的数据表定义之前，先删除原来可能已经存在的（`drop=yes`）。有关这一配置和其他配置选项的更多细节，可以查阅Hibernate Tools参考手册。

Hibernate甚至可以检查现有的数据表，并尝试计算出如何修改schema，才能反映出新映射文件的变化。

在添加了这些内容以后，就可以准备为我们的TRACK表生成数据库schema了。Hibernate可以为实现我们的目标而完成许多奇特的操作，并井然有序地完成处理。我们只需要为一定的消息配置好日志处理，就可以很容易地观察到Hibernate进行的操作过程。为此，我们需要配置Log4j，这是Hibernate所使用的日志处理环境。最简单的配置方法是直接在类路径下放一个log4j.properties文件。我们可以利用现有的prepare target，在Ant复制Hibernate的.properties文件时，顺便将log4j.properties文件从src目录复制到classes目录。

在src目录下创建一个名为log4j.properties的文件，内容如例2-6所示。（一种简单的做法就是从你下载的Hibernate发行包中的doc/tutorial/src目录将这个文件复制出来，因为该文件就是给Hibernate发行包所带的示例使用的。如果你自己输入，则可以跳过注释块的内容，它们只是介绍一些有用的日志功能的其他选项。）

例2-6: log4j.properties日志配置文件

```
###direct log messages to stdout###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE}%5p%c{
1}: %L-%m%n
###direct messages to file hibernate.log###
```

```

#log4j.appender.file=org.apache.log4j.FileAppender
#log4j.appender.file.File=hibernate.log
#log4j.appender.file.layout=org.apache.log4j.PatternLayout
#log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE}%5p%c{1
}: %L-%m%n
###set log levels-for more verbose logging
change 'info' to 'debug'###
log4j.rootLogger=warn, stdout
log4j.logger.org.hibernate=info
#log4j.logger.org.hibernate=debug
###log HQL query parser activity
#log4j.logger.org.hibernate.hql.ast.AST=debug
###log just the SQL
#log4j.logger.org.hibernate.SQL=debug
###log JDBC bind parameters###
log4j.logger.org.hibernate.type=info
#log4j.logger.org.hibernate.type=debug
###log schema export/update###
log4j.logger.org.hibernate.tool.hbm2ddl=debug
###log HQL parse trees
#log4j.logger.org.hibernate.hql=debug
###log cache activity###
#log4j.logger.org.hibernate.cache=debug
###log transaction activity
#log4j.logger.org.hibernate.transaction=debug
###log JDBC resource acquisition
#log4j.logger.org.hibernate.jdbc=debug
###enable the following line if you want to track down
connection###
###leakages when using DriverManagerConnectionProvider###
#log4j.logger.org.hibernate.connection.DriverManagerConnectionPro
vider=trace

```

有了日志配置文件以后，你可能会想编辑build.xml中的codegen目标，使其也依赖于新的prepare目标。这样可以确保无论何时调用codegen，都配置了日志处理，而且Hibernate配置也可以使用，消除了我们第一次使用它时的麻烦。

现在可以制作数据库模式了！在项目目录的命令行下，执行命令ant prepare，接着再执行ant schema。你会看到类似例2-7内容的输出，

同时还会创建classes目录，并在其中生成了各种资源文件；紧接着就会运行模式生成器（schema generator）。

例2-7：使用HSQLDB的嵌入式数据库服务器来建立数据库模式

```
%ant prepare
Buildfile: build.xml
prepare:
[mkdir]Created
dir: /Users/jim/svn/oreilly/hib_dev_2e/current/examples/
ch02/classes
[copy]Copying 3 files
to/Users/jim/svn/oreilly/hib_dev_2e/current/
examples/ch02/classes
BUILD SUCCESSFUL
Total time: 0 seconds
%ant schema
Buildfile: build.xml
prepare:
schema:
[hibernatetool]Executing Hibernate Tool with a Standard
Configuration
[hibernatetool]1.task: hbm2ddl (Generates database schema)
[hibernatetool]22: 38: 21, 858 INFO Environment: 514-Hibernate
3.2.5
[hibernatetool]22: 38: 21, 879 INFO Environment: 532-loaded
properties from reso
urce hibernate.properties: {hibernate.connection.username=sa,
hibernate.connecti
on.password=****,
hibernate.dialect=org.hibernate.dialect.HSQLDialect, hibernate.
connection.shutdown=true, hibernate.connection.url=jdbc: hsqldb:
data/music, hibe
rnative.bytecode.use_reflection_optimizer=false,
hibernate.connection.driver_class
=org.hsqldb.jdbcDriver}
[hibernatetool]22: 38: 21, 897 INFO Environment: 681-Bytecode
provider name: cg
lib
[hibernatetool]22: 38: 21, 930 INFO Environment: 598-using JDK 1.4
java.sql.Time
stamp handling
```

```

[hibernatetool]22: 38: 22, 108 INFO Configuration: 299-Reading
mappings from file
  e: /Users/jim/Documents/Work/OReilly/svn_hibernate/current/example
es/ch02/classes/
  com/oreilly/hh/data/Track.hbm.xml
[hibernatetool]22: 38: 22, 669 INFO HbmBinder: 300-Mapping class:
com.oreilly.hh.
  data.Track->TRACK
[hibernatetool]22: 38: 22, 827 INFO Dialect: 152-Using dialect:
org.hibernate.dialect.HSQLDialect
[hibernatetool]22: 38: 23, 186 INFO SchemaExport: 154-Running
hbm2ddl schema export
[hibernatetool]22: 38: 23, 194 DEBUG SchemaExport: 170-import file
not found: /import.sql
[hibernatetool]22: 38: 23, 197 INFO SchemaExport: 179-exporting
generated schema
to database
[hibernatetool]22: 38: 23, 232 INFO
DriverManagerConnectionProvider: 41-Using Hibernate built-in
connection pool (not for production use!)
[hibernatetool]22: 38: 23, 234 INFO
DriverManagerConnectionProvider: 42-Hibernate connection pool
size: 20
[hibernatetool]22: 38: 23, 241 INFO
DriverManagerConnectionProvider: 45-autocommit mode: false
[hibernatetool]22: 38: 23, 255 INFO
DriverManagerConnectionProvider: 80-using driver: org.hsqldb.
jdbcDriver at URL: jdbc:hsqldb: data/music
[hibernatetool]22: 38: 23, 258 INFO
DriverManagerConnectionProvider: 86-connection properties:
{user=sa, password=****, shutdown=true}
[hibernatetool]drop table TRACK if exists;
[hibernatetool]22: 38: 23, 945 DEBUG SchemaExport: 303-drop
table TRACK if exists;
[hibernatetool]create table TRACK (TRACK_ID integer generated
by default as identity (start with 1) , title varchar (255)
not null, filePath varchar (255) not null, playTime time,
added date, volume smallint not null, primary key (TRACK_ID)
) ;
[hibernatetool]22: 38: 23, 951 DEBUG SchemaExport: 303-create
table TRACK (TRACK_ID integer generated by default as identity
(start with 1) , title varchar (255) not null, filePath
varchar (255) not null, playTime time, added date, volume
smallint not null, primary key (TRACK_ID) ) ;

```

```
ot null, filePath varchar (255) not null, playTime time, added
date, volume small
int not null, primary key (TRACK_ID) );
[hibernatetool]22: 38: 23, 981 INFO SchemaExport: 196-schema
export complete
[hibernatetool]22: 38: 23, 988 INFO
DriverManagerConnectionProvider: 147-cleanin
g up connection pool: jdbc: hsqldb: data/music
BUILD SUCCESSFUL
Total time: 2 seconds
```

在模式导出（`schema export`）部分的末尾，你能够看到Hibernate用于创建TRACK表的真实SQL语句。如果你查看data目录下music.script文件的开头部分，就会发现它已经整合到数据库了。为了采用更友好的（也许是更方便的）方式来查看数据，可以执行`ant db`来启动HSQLDB图形界面，如图2-1所示。

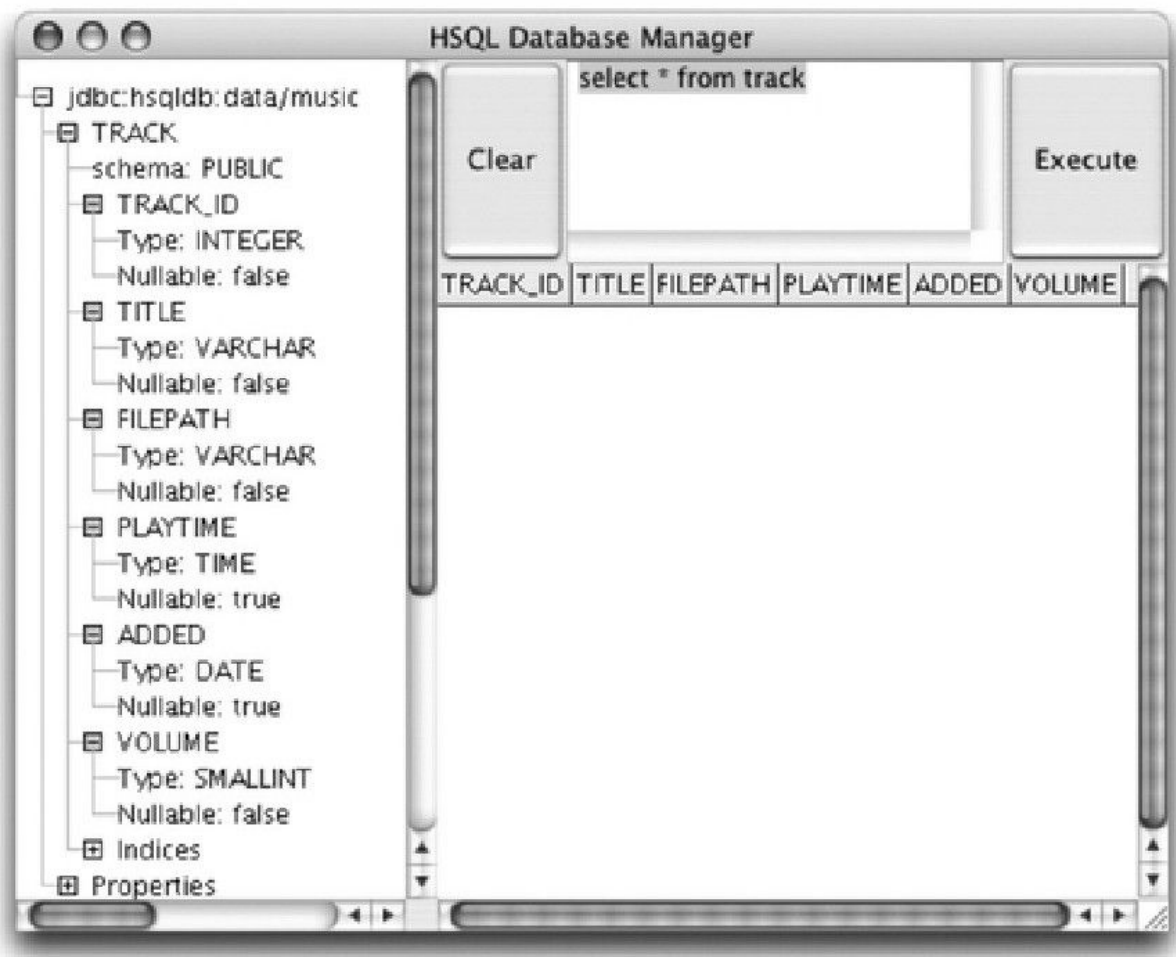


图 2-1 显示新的TRACK表的内容以及一个查询的数据库管理器界面

细心的读者可能会问，既然schema target已经依赖于prepare了，那为什么要调用两次Ant，第一次是运行prepare，第二次是运行schema。这是一种所谓的步步为营法（bootstrapping）的问题，只在第一次创建环境时才会遇到。具体问题是，Ant在启动时会处理属性定义，以决定项目类路径的内容。直到prepare至少运行一次以前，classes目录还不存在，也不会包含hibernate.properties文件，所以在类路径中也不包含这个目录。在prepare运行以后，才会在classes目录生成这个文件，而

`classes`目录也才会包含在类路径中。但是Ant在下一次运行以前，不会重新设置属性定义。所以，如果你试图在第一次Ant运行时就运行`schema`目标，Hibernate就会出现异常，向你报告没有指定一个数据库方言，因为它找不到配置属性文件。这种问题经常会导致令人困惑的尴尬。不过，从现在起，可以安全地直接运行`schema`目标了，并通过它来调用`prepare`，按照需要复制新版本的属性文件，因为它们在Ant开始运行时，就已经在类路径中存在了。

发生了什么事

刚才我们已经能够使用Hibernate创建一个数据表了，以后我们可以将Hibernate为我们创建的Java类实例持久地保存在这个数据表中。我们没有输入任何一行SQL或Java语句！当然，我们的数据库表目前还是空的。来改变它吧！下一章会介绍你可能最想学习的主题：在Java程序中使用Hibernate将Java对象转换为数据库中的数据项，并进行相反的转换。

在深入探讨那种非常酷的任务之前，我们应该再回想一下通过一些XML和`.properties`文件所能做到的诸多事项，这是非常有价值的。希望你已经开始看到Hibernate的强大功能和使用上的便利了。

其他

其他ID生成方法呢？主键（Key）在整个数据库中或全世界都是惟一的吗？Hibernate支持很多为存储在数据库中的对象选择相应的主键的方法。这是由generator标签控制的，如例2-1的第5行所示。在这个示例中，我们告诉Hibernate，对其遇到的数据库类型使用最自然的主键生成方法（native）。其他方法还包括流行的"hi/lo"算法、全局UUID、完全由Java程序代码决定的方法以及其他多种方法。具体细节，可以参阅Hibernate参考手册文档中"Basic O/R Mapping"那一章的"generator"一节。此外，如果内建的各种方法都满足不了你的需要（时常如此），还可以提供你自己的类来做你想做的事情（实现org.hibernate.id.Identifier-Generator接口，再把实现类的名称放在generator标签内）。

如果你想学习一下用Hibernate连接你所更熟悉的数据库的示例，可以先看看第10章，这一章将演示如何用Hibernate处理MySQL数据库。

第3章 驾驭Hibernate

好了，通过前两章的学习，我们已经打牢了基础，定义好对象/关系映射（object/relational mapping）后，又用它创建了相匹配的Java类和数据库表。但这如何应用呢？现在就介绍一下用Hibernate在Java程序代码中处理持久化（persistent）数据是多么的方便。

配置Hibernate

在我们继续学习Hibernate的使用之前，我们需要解决某些妨碍继续前进的问题。在上一章中，我们使用src目录下的hibernate.properties文件来配置Hibernate的JDBC连接。在这一章中，我们将使用Hibernate XML配置文件来配置JDBC连接、SQL方言等各种Hibernate设置。与hibernate.properties文件一样，我们也把这个XML配置文件放在src目录中。将例3-1的内容输入到一个名为hibernate.cfg.xml的文件中，并将其保存在src目录下，再删除原来的hibernate.properties文件。

例3-1：用XML配置Hibernate：hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<! DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
```

```

    <session-factory>
    <!--SQL dialect-->
    <property name="dialect">org.hibernate.dialect.HSQLDialect
</property>❶
    <!--Database connection settings-->❷
    <property name="connection.driver_class">org.hsqldb.jdbcDriver
</property>
    <property name="connection.url">jdbc: hsqldb: data/music
</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="connection.shutdown">>true</property>
    <!--JDBC connection pool (use the built-in one) -->
    <property name="connection.pool_size">1</property>❸
    <!--Enable Hibernate's automatic session context management-->
    <property name="current_session_context_class">thread
</property>
    <!--Disable the second-level cache-->❹
    <property
    name="cache.provider_class">org.hibernate.cache.NoCacheProvider
</property>
    <!--disable batching so HSQLDB will propagate errors
correctly.-->
    <property name="jdbc.batch_size">0</property>❺
    <!--Echo all executed SQL to stdout-->
    <property name="show_sql">>true</property>❻
    <!--List all the mapping documents we're using-->❼
    <mapping resource="com/oreilly/hh/data/Track.hbm.xml"/>
</session-factory>
</hibernate-configuration>

```

从示例中可以看到，hibernate.cfg.xml配置了SQL方言、JDBC参数、连接池（connection pool）以及缓存提供者（cache provider）。这个XML文档还引用了我们在上一章写的映射文档，这样就不需要我们从Java源代码中引用这些映射文档了。稍后将详细介绍这一配置文件的细节：

❶就像第2章的hibernate.properties文件一样，我们在这一行是为使用HSQLDB而定义它的SQL方言。你可能已经注意到property元素的name属性是dialect，类似于.properties文件中的属性的名称（也就是hibernate.dialect）。当使用XML配置文件来配置Hibernate时，其实也是在向Hibernate传递了同样的属性。在XML配置文件中，可以省略属性名称的hibernate.前缀。这一区段（dialect）和下一区段（connection）的配置方法与我们在第2章hibernate.properties文件中的配置是一样的。

❷用于配置Hibernate内部的JDBC连接的属性（connection.driver_class、connection.url、connection.username、connection.password以及connection.shutdown）与例2-4的hibernate.properties中的属性集也一一对应。

❸将connection.pool_size属性设置为1。这意味着Hibernate将创建一个只保持一个连接的JDBC Connection（连接）池。数据库连接池对于需要达到一定规模的大型应用程序来说很重要，不过就本书的目的来说，我们可以放心地将Hibernate配置为使用只有一个JDBC连接的内建连接池。在连接池实现上，Hibernate支持很大的灵活性，可以非常容易地配置Hibernate来使用其他的连接池实现，例如Apache Commons DBCP和C3P0。

④照目前情况，当Hibernate执行实际的持久化操作时，它就会警告我们还没有配置它的二级缓存系统。对于像这样的简单应用程序来说，我们根本不需要二级缓存；所以这一行配置就是要关闭二级缓存，将每个操作立即发送到数据库。

⑤这里，我们是在关闭Hibernate的JDBC批处理功能。虽然这样做会稍微降低一点效率（对于像HSQLDB这样的内存数据库的影响微乎其微），不过为了获取当前HSQLDB触发的错误报告，有必要这样做。当打开批处理模式时，如果批处理中有任何语句出了问题，从HSQLDB中得到的惟一异常将只是BatchUpdateException，告诉你批处理失败。这样错误报告对调试程序几乎没有什么用。HSQLDB的作者说这个问题将在下一个主要发行版本中得以修复；在此之前，当使用HSQLDB时，为了明白问题的来龙去脉，我们就先不使用批处理模式。

⑥show_sql属性是一个在开发和调试Hibernate程序时使用的属性。将show_sql设置为true，就告诉Hibernate要它打印输出对数据库操作时执行的每条语句。如果你不希望在控制台看到打印输出的SQL语句，可以将这个属性设置为false。

⑦最后一部分列出了在我们的项目中使用的映射文档。注意，路径中包含的斜杠字符（“/”）是相对于src目录的。这一路径指明了.hbm.xml文件作为资源在类路径上的位置。通过在这里列出这些文

件，我们就不用在处理映射类的build.xml构建目标中显式说明如何找到它们（稍后可以看到），也不需要像本书的旧版那样在每个例子源代码的main（）方法中加载这些映射文档。把所有映射文档集中在一起是个不错的做法。

除了src目录下的hibernate.cfg.xml文件，还需要修改build.xml以引用这个新的XML配置文件。修改codegen和schema构建目标内的configuration元素，如例3-2中用粗体显示的几行所示。例3-2：修改build.xml，以使用新的Hibernate XML配置文件

```
.....
<!--Generate the java code for all mapping files in our source
tree-->
<target name="codegen"depends="prepare"
description="Generate Java source from the O/R mapping files">
<hibernatetool destdir="${source.root}">
<configuration
configurationfile="${source.root}/hibernate.cfg.xml"/>
<hbm2java/>
</hibernatetool>
</target>
.....
<!--Generate the schemas for all mapping files in our class
tree-->
<target name="schema"depends="prepare"
description="Generate DB schema from the O/R mapping files">
<hibernatetool destdir="${source.root}">
<configuration
configurationfile="${source.root}/hibernate.cfg.xml"/>
<hbm2ddl drop="yes"/>
</hibernatetool>
</target>
.....
```

这两行告诉Hibernate Tools Ant构建任务，到哪里查找Hibernate XML配置文件，这样，Ant任务就可以在这个配置文件中找到它们需要的所有信息（包括正在处理的映射文档；回想一下第2章的做法，我们不得不在configuration元素中显式构建了Ant的fileset元素，以匹配项目源代码树中的所有映射文件）。从现在起，本书的剩余部分将全部使用Hibernate XML配置文件，这样可以更方便地为Hibernate的相关工具传递各种需要的信息。

现在我们已经成功地配置好了Hibernate，下面我们就回到本章的主要任务：生成持久化对象。

创建持久化对象

我们先创建几个新的**Track**实例，再把它们持久化保存到数据库，这样就能看看对象到底是怎样转换成数据库表的行和列的。因为我们完全按照标准的**Hibernate**要求来组织映射文档和配置文件，所以配置**Hibernate**的会话工厂（**session factory**）也就变得相当容易了。

应该怎么做

这里的讨论假设你已经按照第2章的示例，创建好了数据库模式，并生成了**Java**代码。如果你还没有做这些，可以从本书的网站（[\[1\]](#)）下载示例文件，直接跳转到**ch03**目录，使用命令**ant prepare**和**ant codegen**（[\[2\]](#)），再接着执行**ant schema**，就可以自动取回这个示例所需要的**Hibernate**和**HSQLDB**库，并生成**Java**代码和数据库模式。（和其他示例一样，这些命令都应该在**shell**窗口中执行，而当前工作目录就是你的项目目录树的顶级目录，也就是包含**Ant build.xml**文件的地方。）

我们先从一个简单的示范用的**CreateTest**类开始，它包含了必要的导入（**import**）语句，以及一些辅助代码，用于设定**Hibernate**环境，再

创建几个用XML映射文件来进行持久化存储的Track实例。源代码如例3-3所示，它位于src/com/oreilly/hh目录中。

例3-3：数据创建测试，CreateTest.java

```
package com.oreilly.hh;
import org.hibernate.*; ❶
import org.hibernate.cfg.Configuration;
import com.oreilly.hh.data.*;
import java.sql.Time;
import java.util.Date;
/**
 *Create sample data, letting Hibernate persist it for us.
 */
public class CreateTest{
public static void main (String args[]) throws Exception{
//Create a configuration based on the XML file we've put
//in the standard place.
Configuration config=new Configuration () ; ❷
config.configure () ;
//Get the session factory we can use for persistence
SessionFactory sessionFactory=config.buildSessionFactory () ; ❸
//Ask for a session using the JDBC information we've configured
Session session=sessionFactory.openSession () ; ❹
Transaction tx=null;
try{
//Create some data and persist it
tx=session.beginTransaction () ; ❺
Track track=new Track ("Russian Trance",
"vol2/album610/track02.mp3",
Time.valueOf ("00: 03: 30") , new Date () ,
(short) 0) ;
session.save (track) ;
track=new Track ("Video Killed the Radio Star",
"vol2/album611/track12.mp3",
Time.valueOf ("00: 03: 49") , new Date () ,
(short) 0) ;
session.save (track) ;
track=new Track ("Gravity's Angel",
"vol2/album175/track03.mp3",
Time.valueOf ("00: 06: 06") , new Date () ,
(short) 0) ;
session.save (track) ;
```

```
//We're done; make our changes permanent
tx.commit () ; ❹
}catch (Exception e) {
if (tx!=null) {
//Something went wrong; discard all partial changes
tx.rollback () ;
}
throw new Exception ("Transaction failed", e) ;
}finally{
//No matter what, close the session
session.close () ;
}
//Clean up after ourselves
sessionFactory.close () ; ❺
}
}
```

需要对CreateTest.java的第一部分做些解释:

❶我们导入一些有用的Hibernate类（包括Configuration），用它们来建立Hibernate运行环境。此外还需要导入Hibernate根据映射文档生成的所有数据类，这些都在data包中。数据对象中用Time和Date类来代表曲目的播放时间和创建时间戳（timestamp）。在CreateTest中实现的惟一方法是main（）方法，以支持从命令行的调用。

❷当运行这个类时，它首先创建一个Hibernate Configuration对象。由于我们没有告诉Hibernate什么其他信息，所以它默认在类路径的根上查找名为hibernate.cfg.xml的文件。Hibernate会找到我们前面创建的这个配置文件，通过这个配置文件来告诉Hibernate我们正在使用HSQLDB，以及如何找到数据库。这个例子的XML配置文件就包含了

对Track对象的Hibernate Mapping XML文档的引用。调用 `config.configure ()` 就会自动加载Track类的映射文档。

❸为了创建和持久化曲目数据，这就是我们需要的所有配置，这样就为创建SessionFactory做好了准备。该对象的用途是为我们提供会话对象，它是同Hibernate进行交互的主要途径。SessionFactory是线程安全的，在整个应用程序中只需要创建它的一个实例（更准确地说，对于每一个需要提供持久化服务的数据库环境，只需要一个SessionFactory实例；因此大多数应用程序只需要一个这样的实例）。创建会话工厂是一个需要花费相当代价和耗时的操作，所以应该在整个应用程序中共享这个实例。在只包含一个类的应用程序中进行这样的操作显得有些繁琐，不过，参考文档提供了一些在更实现的场景中应该如何应用的好例子。

注意：牢固地理解这些对象的目的和生命周期，值！本书介绍的知识足以让你起步了；你应该继续花些时间阅读参考文档，深入理解各个例子。

❹这一步才到了真正执行持久化的时候，让SessionFactory打开一个会话，这会建立一个到数据库的连接，并提供一个上下文（context）环境，在这个上下文中我们可以创建、获取、处理以及删除持久化对象。只要保持会话为打开状态，就会维护一个到数据库的连接，与会话相关联的持久对象的变化都可以被跟踪；这样，当关闭

会话时，这些变化就可以应用到数据库。从概念上说，你可以把会话认为是持久化对象和数据库之间的一个“大型事务”，它可以包含多个数据库级的事务。不过，和数据库事务一样，在应用程序运行期间长时间地打开**Hibernate session**（例如在等候用户输入时）。在应用程序中一个会话只用于一个特定的、边界有限的操作，例如生成用户界面或者根据用户提交的信息做出相应的变化。而随后的操作则使用一个新的会话。还要注意的，会话对象本身不是线程安全的，所以不能在线程之间共享它们。每个线程需要从会话工厂类中获取它们自己的会话。

有必要深入介绍一下**Hibernate**中映射对象的生命周期，以及它与会话的关系，因为这一术语相当特殊，相关的概念也很重要。一个映射对象（例如我们的**Track**类的一个实例）会在与**Hibernate**相关的两个状态之间回来转换：瞬时状态（**transient**）和持久化状态

（**persistent**）。处于瞬时状态的对象不与任何会话关联。当用**new**（）第一次创建一个**Track**实例时，它就是瞬时状态的；除非告诉**Hibernate**持久化这个瞬时状态的对象，否则当应用程序结束时，这个对象就会永远消失。

将一个瞬时状态的映射对象传递给会话的**save**（）方法，就会持久化保存这个对象，这样，在**Java VM**结束以后，这个数据对象还能够存在，直到以后显式地删除它。如果你已经有了一个持久化对象，并对

它调用会话的`delete()`方法，这个对象就会再回到瞬时状态。虽然这个对象在应用程序中仍然作为一个实例而存在，但它不会持久保存起来，除非你改变了主意，要再保存它一次。另一方面，如果你还没有删除这个对象（所以它仍然处于持久化状态），当修改这个对象后，为了让对象的变化得以反映到数据库中，并不需要再显式地保存它一次。**Hibernate**会自动跟踪对任何持久化对象作出的修改，并在适当的时机将这些变化刷新写入到数据库中。当关闭会话时，任何延迟的变化都会被保存到数据库中。

注意：坚持一下，我们很快就回到例子的介绍！

当在已经关闭的会话中使用持久化对象，例如，当运行完一个查询，找到所有匹配某条件的实体（本章稍后的“检索持久化对象”一节将介绍如何做到这一点）以后，处理这些实体的状态就涉及一个重要但又微妙的要点。如前所述，保持会话打开的时间最好不要超过执行数据库操作的必要时间，所以在查询完成以后，就应该马上关闭会话。如果这时再处理已经加载的映射对象，会怎么样？嗯，当会话打开时，这些对象确实是持久化对象，但是它们现在不再与任何活动的会话相关联了（在这个例子中，是因为关闭了会话），所以它们就不再是持久化对象了。不过，这并不意味着它们代表的数据库数据也不存在了；相反，如果再次运行查询（假设这时没有人修改过数据），还是能够取回同样的一组对象。这只是意味着：数据对象的状态

态在虚拟机和数据库之间当前没有通信，它们处于脱管状态

（**detached**）。完全有理由可以继续使用这种对象。如果以后需要修改这些对象，还想把这些修改持久保存，你可以打开一个新的会话，用它来保存修改过的对象。因为每个实体都有一个惟一的ID，在新的会话中，**Hibernate**没有问题地可以知道如何把处于瞬时状态的对象链接到正确的持久化对象。

当然，对脱管对象信息的修改都要由具体的数据库环境作为支持，所以你需要考虑应用程序级的数据完整性约束。可能需要设计某种高级的锁定或版本化协议来支持脱管对象的管理。虽然**Hibernate**为这一任务也提供了一定帮助，但是设计和实现细节还得由你负责。参考手册强烈推荐使用一个**version**字段，而且也有多种办法可供选用。

⑤有了这些概念和术语，这个例子的其他部分就很容易理解了。我们用打开的会话建立了一个数据库事务，在这个事务内部，又创建了一些包含示例数据的**Track**实例，并在会话中进行保存，这样就把它由瞬时状态的实例转变为持久化的实体。

⑥最后，我们提交事务，自动地（作为一个单独的、不可分割的单元）将所有数据库修改持久化。环绕着所有这些代码周围的**try/catch/finally**块演示了在进行事务处理时一种重要而且有用的习惯用法。如果操作期间发生了任何错误，**catch**块就会回滚（**roll back**）事务，再抛出异常。在**finally**部分中会关闭会话，以确保无论我们是成功

提交事务后沿着“愉快的小路”正常退出，还是因为导致回滚的异常而退出，最终都可以关闭会话。

⑦在方法最后，我们也关闭了会话工厂本身。这是应用程序中“优雅地关闭”（graceful shutdown）部分应该进行的操作。在Web应用环境中，这应该是一定的生命周期事件处理器。在这个简单的例子中，当main（）方法返回时，应用程序就正在结束。

现在一切就绪，通知Ant如何编译和运行这个测试程序就更简单了。将例3-4所示的构建目标添加到build.xml末尾的</project> 关闭标签之前。

例3-4：用于编译所有Java源代码，并调用数据创建测试的Ant构建目标

```
<!--Compile the java source of the project-->
<target name="compile"depends="prepare"①
description="Compiles all Java classes">
<javac srcdir="${source.root}"
destdir="${class.root}"
debug="on"
optimize="off"
deprecation="on">
<classpath refid="project.class.path"/>
</javac>
</target>
<target name="ctest"description="Creates and persists some
sample data"
depends="compile">②
<java classname="com.oreilly.hh.CreateTest"fork="true">
<classpath refid="project.class.path"/>
</java>
</target>
```

❶命名得体的编译构建目标使用内建的javac构建任务，将src目录中的所有Java源文件编译到classes目录中。幸好，这个构建任务也支持我们已经建立的项目类路径，所以编译器能够找到我们正在使用的所有依赖库。构建目标定义中的depends=prepare属性是告诉Ant在运行编译构建目标以前，必须先运行prepare。Ant负责管理依赖关系，当构建具有依赖关系的多个目标时，能够以正确的顺序执行，每个依赖只执行一次，即便多个构建目标都引用了同一个依赖。

如果你习惯使用shell脚本来编译很多Java源代码，那么可能会对这么快的编译速度感到吃惊。Ant调用的是它自己使用的虚拟机内部的Java编译器，所以没有针对每个编译的处理启动延迟。

❷ctest构建目标使用编译来确保已经构建好了所有类文件，接着再创建一个新的Java虚拟机来运行我们的CreateTest类。

好了，我们可以创建一些数据了！例3-5显示了调用新的ctest构建目标的结果。ctest要依赖于compile构建目标，这样就能确保在使用之前CreateTest类会先编译好。ctest本身的输出结果会显示Hibernate所发出的日志信息（建立环境和映射数据以及数据库连接的关闭）。

例3-5：调用CreateTest类

```
%ant ctest
prepare:
compile:
```

```

[javac]Compiling 2 source files
to/Users/jim/svn/oreilly/hib_dev_2e/
current/examples/ch03/classes
ctest:
[java]00: 21: 45, 833 INFO Environment: 514-Hibernate 3.2.5
[java]00: 21: 45, 852 INFO Environment: 547-hibernate.properties
not found
[java]00: 21: 45, 864 INFO Environment: 681-Bytecode provider
name: cglib
[java]00: 21: 45, 875 INFO Environment: 598-using JDK 1.4
java.sql.Timestamp
p handling
[java]00: 21: 46, 032 INFO Configuration: 1426-configuring from
resource: /
hibernate.cfg.xml
[java]00: 21: 46, 034 INFO Configuration: 1403-Configuration
resource: /hib
ernate.cfg.xml
[java]00: 21: 46, 302 INFO Configuration: 553-Reading mappings
from resource
e: com/oreilly/hh/data/Track.hbm.xml
[java]00: 21: 46, 605 INFO HbmBinder: 300-Mapping class:
com.oreilly.hh.data
a.Track->TRACK
[java]00: 21: 46, 678 INFO Configuration: 1541-Configured
SessionFactory: null
[java]00: 21: 46, 860 INFO DriverManagerConnectionProvider: 41-
Using Hibernate
ate built-in connection pool (not for production use!)
[java]00: 21: 46, 862 INFO DriverManagerConnectionProvider: 42-
Hibernate connection
nnection pool size: 1
[java]00: 21: 46, 864 INFO DriverManagerConnectionProvider: 45-
autocommit mode: false
[java]00: 21: 46, 879 INFO DriverManagerConnectionProvider: 80-
using driver
: org.hsqldb.jdbcDriver at URL: jdbc:hsqldb: data/music
[java]00: 21: 46, 891 INFO DriverManagerConnectionProvider: 86-
connection properties: {user=sa, password=****, shutdown=true}
[java]00: 21: 47, 533 INFO SettingsFactory: 89-RDBMS: HSQL
Database Engine,
version: 1.8.0
[java]00: 21: 47, 538 INFO SettingsFactory: 90-JDBC driver: HSQL
Database Engine Driver, version: 1.8.0

```

```

    [java]00: 21: 47, 613 INFO Dialect: 152-Using dialect:
org.hibernate.dialect
    t.HSQLDialect
    [java]00: 21: 47, 638 INFO TransactionFactoryFactory: 31-Using
default tran
    saction strategy (direct JDBC transactions)
    [java]00: 21: 47, 646 INFO TransactionManagerLookupFactory: 33-No
Transacti
    onManagerLookup configured (in JTA environment, use of read-write
or transaction
    al second-level cache is not recommended)
    [java]00: 21: 47, 649 INFO SettingsFactory: 143-Automatic flush
during befo
    reCompletion () : disabled
    [java]00: 21: 47, 650 INFO SettingsFactory: 147-Automatic session
close at
    end of transaction: disabled
    [java]00: 21: 47, 657 INFO SettingsFactory: 154-JDBC batch size:
15
    [java]00: 21: 47, 659 INFO SettingsFactory: 157-JDBC batch updates
for vers
    ioned data: disabled
    [java]00: 21: 47, 664 INFO SettingsFactory: 162-Scrollable result
sets: ena
    bled
    [java]00: 21: 47, 666 INFO SettingsFactory: 170-JDBC3
getGeneratedKeys () : d
    isabled
    [java]00: 21: 47, 668 INFO SettingsFactory: 178-Connection release
mode: au
    to
    [java]00: 21: 47, 671 INFO SettingsFactory: 205-Default batch
fetch size: 1
    [java]00: 21: 47, 678 INFO SettingsFactory: 209-Generate SQL with
comments:
    disabled
    [java]00: 21: 47, 680 INFO SettingsFactory: 213-Order SQL updates
by primar
    y key: disabled
    [java]00: 21: 47, 681 INFO SettingsFactory: 217-Order SQL inserts
for batch
    ing: disabled
    [java]00: 21: 47, 684 INFO SettingsFactory: 386-Query translator:
org.hiber
    nate.hql.ast.ASTQueryTranslatorFactory
    [java]00: 21: 47, 690 INFO ASTQueryTranslatorFactory: 24-Using
ASTQueryTran
    slatorFactory

```

```

    [java]00: 21: 47, 694 INFO SettingsFactory: 225-Query language
substitution
    s: {}
    [java]00: 21: 47, 695 INFO SettingsFactory: 230-JPA-QL strict
compliance: d
    isabled
    [java]00: 21: 47, 702 INFO SettingsFactory: 235-Second-level
cache: enabled
    [java]00: 21: 47, 704 INFO SettingsFactory: 239-Query cache:
disabled
    [java]00: 21: 47, 706 INFO SettingsFactory: 373-Cache provider:
org.hibernate
    te.cache.NoCacheProvider
    [java]00: 21: 47, 707 INFO SettingsFactory: 254-Optimize cache for
minimal
    puts: disabled
    [java]00: 21: 47, 709 INFO SettingsFactory: 263-Structured second-
level cac
    he entries: disabled
    [java]00: 21: 47, 724 INFO SettingsFactory: 283-Echoing all SQL to
stdout
    [java]00: 21: 47, 731 INFO SettingsFactory: 290-Statistics:
disabled
    [java]00: 21: 47, 732 INFO SettingsFactory: 294-Deleted entity
synthetic id
    entifier rollback: disabled
    [java]00: 21: 47, 734 INFO SettingsFactory: 309-Default entity-
mode: pojo
    [java]00: 21: 47, 735 INFO SettingsFactory: 313-Named query
checking: enab
    led
    [java]00: 21: 47, 838 INFO SessionFactoryImpl: 161-building
session factory
    [java]00: 21: 48, 464 INFO SessionFactoryObjectFactory: 82-Not
binding fact
    ory to JNDI, no JNDI name configured
    [java]Hibernate: insert into TRACK (TRACK_ID, title, filePath,
playTime, a
    dded, volume) values (null, ?, ?, ?, ?, ?)
    [java]Hibernate: call identity ()
    [java]Hibernate: insert into TRACK (TRACK_ID, title, filePath,
playTime, a
    dded, volume) values (null, ?, ?, ?, ?, ?)
    [java]Hibernate: call identity ()
    [java]Hibernate: insert into TRACK (TRACK_ID, title, filePath,
playTime, a
    dded, volume) values (null, ?, ?, ?, ?, ?)
    [java]Hibernate: call identity ()

```

```
[java]00: 21: 49, 365 INFO SessionFactoryImpl: 769-closing
[java]00: 21: 49, 369 INFO DriverManagerConnectionProvider: 147-
cleaning up
connection pool: jdbc: hsqldb: data/music
BUILD SUCCESSFUL
Total time: 2 seconds
```

发生了什么事

如果你查看Hibernate打印输出的所有消息（因为我们打开了日志的"info"级别的输出标志），你可以看到我们的测试类会启动Hibernate，加载Track类的映射信息，打开一个持久会话（persistence session）以连接相关的HSQLDB数据库，再创建一些实例，并用会话将它们持久化保存到TRACK数据库表中。接着，再关闭这个会话和数据库连接，以确保数据正确地完成存储。

运行完这个测试以后，你可以用ant db看一看数据库的内容。现在，你应该可以在TRACK表中看到三条记录，如图3-1所示（在窗口顶部的文本框中输入查询语句，点击"Execute"按钮。也可以选择菜单栏中的"Command" → "Select"，会得到一个SQL命令的框架和语法说明文档）。



图 3-1 持久保存到TRACK表中的测试数据

现在，我们停下来一会儿，反思一个事实——我们没有编写任何连接数据库或执行SQL命令的代码。再回想上一节，我们甚至也不必亲自创建数据库表和封装数据的Track对象。但是，图3-1中查询输出显示的那些整整齐齐的数据表明，我们简短的测试程序确实已经创建了Java对象，并正确地进行了持久化处理。希望你可以因此而认同，作为一种持久化服务，**Hibernate**真的功能强大、使用方便。作为一种免费的、轻型的工具，**Hibernate**确实为我们完成了很多工作，这一切又是那么的快捷而简单！

如果你以前直接用过JDBC，尤其是当你还不太熟悉它时，你可能习惯使用数据库驱动程序的"auto-commit"（自动提交）模式，而不是使用数据库事务处理（transaction）。**Hibernate**同样认为这是构造应用程序的一种错误方法，“自动提交”模式惟一有意义的使用场合就是供

人使用的数据库控制台环境。所以，在Hibernate应用中，持久化操作总是需要使用事务处理。

如第1章所述，你可以在data目录下的music.script文件中直接查看用于创建数据的SQL语句，如例3-6所示。

例3-6: 查看原始的数据库脚本文件

```
%cat data/music.script
CREATE SCHEMA PUBLIC AUTHORIZATION DBA
CREATE MEMORY TABLE TRACK (TRACK_ID INTEGER GENERATED BY DEFAULT
AS IDENTITY (STAR
T WITH 1) NOT NULL PRIMARY KEY, TITLE VARCHAR (255) NOT NULL,
FILEPATH VARCHAR (255)
NOT NULL, PLAYTIME TIME, ADDED DATE, VOLUME SMALLINT NOT NULL)
ALTER TABLE TRACK ALTER COLUMN TRACK_ID RESTART WITH 4
CREATE USER SA PASSWORD""
GRANT DBA TO SA
SET WRITE_DELAY 10
SET SCHEMA PUBLIC
INSERT INTO TRACK VALUES (1, 'Russian
Trance', 'vol2/album610/track02.mp3', '00: 03: 3
0', '2007-06-17', 0)
INSERT INTO TRACK VALUES (2, 'Video Killed the Radio
Star', 'vol2/album611/track12.
mp3', '00: 03: 49', '2007-06-17', 0)
INSERT INTO TRACK VALUES (3, 'Gravity's
Angel', 'vol2/album175/track03.mp3', '00: 06
: 06', '2007-06-17', 0)
```

最后三条语句是我们的TRACK表的数据行。脚本的最前面包含了当创建新的数据库时，默认使用的模式和用户名。（当然，在实际应用环境中，你可能需要修改这些身份验证信息，除非数据库只供内存访问（in-memory access）。）

注意：想多学点HSQLDB？我们支持你！

其他

对象和其他对象之间的关系呢？对象集合（collection）呢？没错，有些情况会让持久化处理更具挑战性（如果做得好的话，就相当有价值）。Hibernate能妥善处理这种关联性。事实上，我们不需要为此花费什么力气。在第4章将会讨论这一点。就目前而言，让我们先看看如何将先前会话中持久保存的对象取出来。

[1] <http://www.oreilly.com/catalog/9780596517724/>.

[2] 虽然codegen构建目标要依赖prepare构建目标，但第一次处理示例目录时，你需要先显式地运行prepare以创建正确的类路径结构，这样Ant以后才可以正常工作，如第2章的2.3节所述。

检索持久化对象

现在，是该来个180度大转弯，看看如何从数据库加载数据到Java对象中。

使用Hibernate查询语言（Hibernate Query Language, HQL），就可以面向对象的方法来检索映射到数据库表的内容。这些数据可以是以前的会话中保存的持久化对象，也可以是完全来自于应用程序代码以外的数据。

应该怎么做

例3-7演示了一个程序，对我们刚才创建的测试数据进行简单的查询。整体的结构看起来应该非常熟悉，因为所有的Hibernate设置都和上一个程序相同。

例3-7：数据检索测试QueryTest.java

```
package com.oreilly.hh;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import com.oreilly.hh.data.*;
import java.sql.Time;
import java.util.*;
/**
 *Retrieve data as objects
 */
```

```

public class QueryTest{
/**
 *Retrieve any tracks that fit in the specified amount of time.
 *
 *@param length the maximum playing time for tracks to be
returned.
 *@param session the Hibernate session that can retrieve data.
 *@return a list of{@link Track}s meeting the length restriction.
 */
    public static List tracksNoLongerThan (Time length, Session
session) {❶
        Query query=session.createQuery ("from Track as track"+
        "where track.playTime<=?") ;
        query.setParameter (0, length, Hibernate.TIME) ;
        return query.list () ;
    }
/**
 *Look up and print some tracks when invoked from the command
line.
 */
    public static void main (String args[]) throws Exception{
        //Create a configuration based on the properties file we've put
        //in the standard place.
        Configuration config=new Configuration () ;
        config.configure () ;
        //Get the session factory we can use for persistence
        SessionFactory sessionFactory=config.buildSessionFactory () ;
        //Ask for a session using the JDBC information we've configured
        Session session=sessionFactory.openSession () ;
        try{❷
            //Print the tracks that will fit in five minutes
            List tracks=tracksNoLongerThan (Time.valueOf ("00: 05: 00") ,
            session) ;
            for (ListIterator iter=tracks.listIterator () ;
            iter.hasNext () ; ) {
                Track aTrack= (Track) iter.next () ;
                System.out.println ("Track: \""+aTrack.getTitle () +
                "\", "+aTrack.getPlayTime () ) ;
            }
        }finally{
            //No matter what, close the session
            session.close () ;
        }
        //Clean up after ourselves
        sessionFactory.close () ;
    }
}

```

同样，如例3-8所示，在build.xml的末尾（在project的关闭标签以前）添加一个构建目标，以运行这个测试。

例3-8：调用查询测试的Ant构建目标

```
<target name="qtest" description="Run a simple Hibernate query"
depends="compile">
<java classname="com.oreilly.hh.QueryTest" fork="true">
<classpath refid="project.class.path"/>
</java>
</target>
```

准备好以后，只要输入ant qtest，就可以检索出数据并显示出来，其结果如例3-9所示。为了节省输出结果占据的空间，我们编辑log4j.properties文件，将所有"info"级别的信息输出都关掉，因为这些信息和前一个例子没有差别。你可以修改一下这一行：

```
log4j.logger.org.hibernate=info
将info替换成warn:
log4j.logger.org.hibernate=warn
例3-9：运行查询测试
%ant qtest
Buildfile: build.xml
prepare:
[copy]Copying 1 file to/Users/jim/svn/oreilly/hib_dev_2e/current
/examples/ch03/classes
compile:
[javac]Compiling 1 source file
to/Users/jim/svn/oreilly/hib_dev_2e
/current/examples/ch03/classes
qtest:
[java]Hibernate: select track0_.TRACK_ID as TRACK1_0_,
track0_.title as ti
tle0_, track0_.filePath as filePath0_, track0_.playTime as
playTime0_, track0_.a
```

```
        dded as added0_, track0_.volume as volume0_from TRACK
track0_where track0_.playTime<=?
[java]Track: "Russian Trance", 00: 03: 30
[java]Track: "Video Killed the Radio Star", 00: 03: 49
BUILD SUCCESSFUL
Total time: 2 seconds
```

发生了什么事

❶我们先是定义了一个工具方法：`tracksNoLongerThan()`，由它执行真正的Hibernate查询，取回播放时间小于或等于参数指定的值的任何曲目。注意HQL（Hibernate根据SQL独创的查询语言）支持参数占位符，非常像JDBC中的`PreparedStatement`。而且，与之类似的是，使用参数占位符更适合于通过字符串处理将所有查询集中在一起（尤其是这样可以免受SQL注入的攻击）。不过，你将看到，Hibernate提供了在Java中更好的使用查询的方法。

查询本身看起来有些古怪。它以`from`作为开始，而不是你可能期待的像`select something`之类的语句。虽然你确实可以使用与标准SQL更加类似的格式，而且当需要在查询中从一个对象提取出单独的属性时，也只能这么做；如果你想获取整个对象，就可以使用这种更简洁的语法。

还要注意的，查询是按照映射的Java对象和属性来表达的，而不是数据表和列。在这个例子中这一区别还不明显，因为对象和数据

表的名称都相同，属性和列的名称也都相同，但查询确实是按照对象和属性来表达的。保持二者的名称一致是一种相当自然的选择，如果使用**Hibernate**来生成数据库模式和数据对象，结果也总是这样，除非你明确地告诉**Hibernate**使用不同的列名称。

当你使用的是以前就有的数据库和对象时，就应该特别留意**HQL**查询引用的是对象属性，而不是数据库表的列。

此外，就像在**SQL**中一样，可以为数据库表和列起其他的别名。在**HQL**中，也可以为类起别名，以方便选择它们的属性或增加约束条件。当然，在这个简单的例子中不会看到这种用法，但是如果你深入研究附录E中的内容，肯定会遇到这种用法。

❷这个程序的其他部分看起来可能和上一个例子类似。这里我们简化了**try**块的处理，因为没有改变任何数据，所以就不需要显式地访问事务。我们仍旧使用**try**块，这样就能够有一个**finally**子句，以清晰地关闭会话。代码体本身很简单，调用我们的查询方法以请求播放时间小于或等于5分钟的所有曲目，接着再遍历结果中的**Track**对象，分别打印它们的标题和播放时间。

既然我们已经关掉了**Hibernate**内部"info"级别的日志输出，那么我们在**hibernate.cfg.xml**中配置的**SQL**调试输出就更容易定位了。输出中"qtest:"部分的第1行并不是我们自己在**QueryTest.java**中编写的，我

们看到的SQL语句是Hibernate生成的，以实现我们请求的HQL查询。这些内幕信息很有趣吧！如果你对这些信息也不感兴趣，则可以将show_sql属性设置为false，就不会看到它们了。

其他

如果你已经对数据创建脚本生成的数据进行了修改，现在又想清空数据库，以一种“干净的状态”（clean slate）来进行测试，那么你需要做的就是再次运行ant schema命令。这样会将原有的TRACK表完全删除，并重新创建新的TRACK表，使其处于最原始的空状态。除非你真的需要这样，否则不要轻易这样做！

如果你想选择性地删除一些数据，则或者通过HSQLDB UI（ant db）里的SQL命令；或者先进行一定的查询，检索回你想删除的对象。在取得持久化对象的引用以后，可以将该对象的引用传递给Session的delete（）方法，这样就可以将它从数据库中删除了：

```
session.delete (aTrack) ;
```

直到aTrack变量超出其作用域范围（scope）或者重新赋值以前，你的程序还至少有一个指向这个被删除对象的引用。因此，就概念上而言，理解delete（）方法最简单的方式就是将其视为把一个持久化对象（persistent object）转变为一个瞬时对象（transient object）。

删除对象的另外一种方法就编写一条可以匹配多个对象的HQL删除查询语句。这样可以一次性删除多个持久化对象，而不管这些对象是否在内存中，还不用自己写循环。除了使用ant schema命令，改用Java方法，以较“温和”的手法来清除掉所有曲目时，就可以像这样写：

```
Query query=session.createQuery ("delete from Track");  
query.executeUpdate ();
```

不要忘了，无论采用哪种方法，都得将数据处理的代码放在Hibernate事务处理以内，如果想让修改的内容“固定”下来，就得提交（commit）该事务。

建立查询的更好方法

如本章前面所述，HQL让你不必使用JDBC风格的查询占位符，就能方便地将参数整合到查询中。可以使用命名参数（`named parameter`）和命名查询（`named query`）来让程序更易于阅读和维护。

为何在意

命名参数之所以可以让代码更容易理解，是因为不管在查询本身内部，还是在建立查询的Java代码内部，都清晰地表达了参数的意图。这种自描述性（`self-documenting`）的本质就是命名参数的价值所在，同时也减少了引发错误的潜在可能，因为使用命名参数时，你不用计算SQL语句中逗号和问号的个数。可以在一个单独的查询中多次使用相同的参数，这样在一定程度上也可以提高程序的性能。

命名查询可以将查询完全从Java代码中分离出来。将查询语句和Java源代码分离开，会使其更易于阅读和编辑，因为查询语句不再是原来一大堆跨越多行的Java字符串序列，且交织着大量的问号、反斜线以及其他Java标点符号。第一次输入这样的语句是相当麻烦的，但是如果你需要对这种嵌入在程序中的查询做重要的修改时，就得四处移动问号和加号，尽可能让语句行再次在适当的位置断开。

应该怎么做

对于Hibernate而言，这些能力的关键就是Query接口。我们在例3-7中就已经使用过这个接口，因为从Hibernate 3开始，Query接口是唯一Hibernate不反对使用（nondeprecated）的执行查询的接口。所以，和本节介绍的其他功能相比，现在更容易了。

我们先修改查询语句，使用命名参数，如例3-10所示。（对于这种只有单一参数的查询语句而言，这并不是什么难事，但宝贵的是从现在起就养成正确的习惯。当你开始为实际的项目打造一大堆难以看清的查询语句时，你会很感激这个习惯的！）

例3-10：修改查询语句，改用命名参数

```
public static List tracksNoLongerThan (Time length, Session
session) {
    Query query=session.createQuery ("from Track as track"+
    "where track.playTime<=: length") ;
    query.setTime ("length", length) ;
    return query.list () ;
}
```

在查询语句内部，命名参数通过在它们的名称前面加一个冒号“:”作为标识。此处，我们将“?”号替换为“:length”。会话对象提供了一个createQuery () 方法，可以返回一个Query接口的实现，以供我们使用。为了设置命名参数的值，Query提供了一整套类型安全的方法。

这里，我们传递的是一个**Time**类型的值，所以我们使用了**setTime**（）方法。即使在如此简单的情况下，和原来的查询语句相比，使用命名参数以后的语法更为自然和方便阅读。如果我们原来传递参数使用的是值和类型的匿名数组（**anonymous array**）（必需传递多个参数），这样的改进就显得更重要了。此外，我们又多加了一层编译时（**compile-time**）的类型检查，这总是很受开发人员欢迎的调整。

运行这一版本的程序产生的结果和原来的程序完全相同。

那么，我们怎么将查询语句文本放到**Java**源代码以外呢？同样，这个查询太简单了，以至于这样做的需要不像在实际项目中那样令人印象深刻。但是，这是处理查询语句的最佳方法，所以开始练习吧！就像你预料的那样，我们存放查询语句的地方就是映射文档内部。例3-11显示了映射文档中的查询语句。我们必须使用有点笨重的**CDATA**结构，因为查询语句中可能会包含一些破坏**XML**解析器处理的字符（例如，像“<”这样的字符）。

例3-11：映射文档中的查询语句

```
<query name="com.oreilly.hh.tracksNoLongerThan">
  <![CDATA[
    from Track as track
    where track.playTime<=: length
  ]]>
</query>
```

将这些内容放在Track.hbm.xml中类定义的闭合标签（</class>）之后（就在</hibernate-mapping>那一行的前面）。然后，对QueryTest.java做最后一次修改，如例3-12所示。同样地，程序产生的结果和最初版本完全相同，只是现在组织得更好。如果我们需要使用更复杂的查询，有这些基础已经相当好了。

例3-12：查询方法的最终版本

```
public static List tracksNoLongerThan (Time length, Session
session) {
    Query query=session.getNamedQuery (
        "com.oreilly.hh.tracksNoLongerThan");
    query.setTime ("length", length);
    return query.list ();
}
```

除了这里探讨的内容以外，Query接口还有很多其他有用的功能。可以使用接口控制要取回多少条记录（并可以指定特定的记录行）。如果JDBC驱动程序支持可滚动的（scrollable）ResultSet，通过Query接口也可以使用这一功能。相关更多的细节，可以查阅JavaDoc和Hibernate的参考手册。

其他

完全不使用类SQL语言？或者深入HQL，探索更为复杂的查询？这两种方法都是本书后面将要介绍的内容。

第8章会讨论条件查询（**criteria query**）。条件查询是一种很有趣的机制，可以让你用普通的**Java API**表达出想要对实体施加的限制条件。这样就可以构建**Java**对象来表示你想要找到的数据，对于非数据库专家来说，这种方法更容易理解。这种方法还可以让你利用**IDE**的代码完成（**code completion**）功能作为一种辅助编辑方法，甚至可以提供编译时的语法检查。此外，**Hibernate**还支持一种“按照例子来查询”（**query by example**）的方法，就是要找什么对象，先提供与之类似的对象例子。这些对实现应用程序的查询搜索接口都很有帮助。

想多看点**HQL**的**SQL**老手可以跳到第9章，那一章将讨论**HQL**的更多其他能力和独特功能。就目前而言，接下来我们要继续探索对象/关系映射中如何处理对象之间的相互关系，这在任何稍复杂的实际应用中都会遇到。

第4章 集合与关联

不，这些不是关于理论的东西。我们已经看到让单独的对象进出数据库是多么容易，现在应该来看看如何处理对象之间的分组和关系。令人高兴的是，这也没什么难的。

集合的映射

在任何真实的应用程序中，总要管理对象的列表或分组。Java提供了一套健壮而功能丰富的类来帮助实现这些应用：集合（Collection）工具。为了将数据库关系映射到集合中，Hibernate也提供了一些很自然的实现方法。而且它们的使用通常也非常方便。不过，你得注意Java集合和Hibernate集合在语义上的一些差别，当然，这些差别很小。事实上最大的差别是Java集合没有提供"bag"（包）的定义，这可能多少会让一些经验丰富的数据库设计师感到失望。这一缺陷并不是Hibernate的错，Hibernate甚至也做一些努力，以作为解决这一问题的变通方案（work around）。

注意：Bag很像Set，只不过相同的值可以多次出现。

这些抽象概念就点到为止！Hibernate参考手册对整个bag问题已经做了详尽的讨论，所以我们在这里不做介绍，直接看一个集合映射的

示例（就此示例而言，关系数据库和Java模型配合得相当好）。在第2章Track示例的基础上继续构建新的功能，将曲目分组成专辑（album）看起来似乎很自然，但对于学习来说，这并不是最简单的起点。因为组成专辑需要牵涉到记录其他额外的信息，例如曲目是从哪张唱片来的（对于那些包含数张唱片的专辑而言），以及其他类似的细节信息。所以，我们先把艺人（artist）的信息加进数据库吧。

注意：和以前一样，这些示例假设你已经按照前几章的步骤做好了基础工作。如果还没有，可以下载示例的源代码作为起点。

要记录的艺人信息相当简单，至少刚开始是这样的。先从艺人的姓名开始。可以为每个曲目分配一组艺人，这样，我们就知道应该找谁表示感谢或不满，还可以找出你喜欢的某位艺人的所有曲目。（允许为一个曲目分配多位艺人是相当重要的，但很少有音乐管理程序做到了这一点。新增加一个链接以记录曲目的作者的工作就留给读者，以作为读者理解这个示例之后的练习吧。）

应该怎么做

就目前而言，我们的Artist类只需要一个name属性就足够了（当然，还有它的主键属性（id））。为它建立映射文档也相当容易。在Track映射文档所在的目录中创建一个名为Artist.hbm.xml的文件，其内容如例4-1所示。

例4-1: Artist类的映射文档

```
<?xml version="1.0"?>
<! DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.oreilly.hh.data.Artist"table="ARTIST">
<meta attribute="class-description">
Represents an artist who is associated with a track or album.
@author Jim Elliott (with help from Hibernate)
</meta>
<id name="id"type="int"column="ARTIST_ID">
<meta attribute="scope-set">protected</meta>
<generator class="native"/>
</id>
<property name="name"type="string">❶
<meta attribute="use-in-tostring">true</meta>
<column name="NAME"not-
null="true"unique="true"index="ARTIST_NAME"/>
</property>
<set name="tracks"table="TRACK_ARTISTS"inverse="true">❷
<meta attribute="field-description">Tracks by this artist</meta
>❸
<key column="ARTIST_ID"/>
<many-to-many
class="com.oreilly.hh.data.Track"column="TRACK_ID"/>
</set>
</class>
</hibernate-mapping>
```

❶我们对name属性的映射引入了一对限制标签，分别用于配置代码生成和模式生成过程。use-in-tostring这个meta标签会让生成的Java类包含一个定制的toString ()方法，用于在输出时显示艺人的名字，以及神秘的散列码（hash code），以辅助调试（生成结果如例4-4底部所示）。扩展column的各属性，使其成为一个更加完整的标签，让我们

对列属性进行更细粒度的控制。在这个例子中，我们用这个标签增加了一个索引，可以提高按艺人名字查询和排序的效率。

❷注意，在这个文件中我们可以很自然地表达一个艺人与一个或多个曲目关联的事实。这段映射告诉Hibernate，为Artist类增加一个名为tracks的属性，它的类型是java.util.Set的一个实现。这会使用一个新的名为TRACK_ARTISTS的表来为该Artist链接由它负责的Track对象。属性inverse=true稍后在例4-3的讨论中再详细解释这种关联的双向性的本质。我们刚才提到的TRACK_ARTISTS表将包含两列：TRACK_ID和ARTIST_ID。在这个表中出现的每一行都意味着指定的Artist对象和指定的Track对象有一定的关系。将这种关联信息单独保存在它自己的表中，这样就不会限制多少个曲目可以链接到一个特定的艺人，也不会限制多少个艺人可以关联到一个曲目。这就是所谓的“多对多”关联。

另一方面，由于这些关联信息保存在一个单独的表中，要获取关于艺人或曲目的任何有意义的信息，就必须执行一个连接查询。这也就是为什么称这样的表为“连接表”（join table）的原因。它们的所有目的就是为了连接其他的表。

最后要注意的是，不像我们用数据库模式定义建立的其他表，TRACK_ARTISTS表没有任何相应的Java对象。它只是用于实现Artist和Track对象之间的链接，体现为Artist的tracks属性。

❸除了普通的值类型字段外，`field-description meta`标签也可以为集合和关联提供JavaDoc描述。当字段名称不能完全自动文档化（`self-documenting`）时，这种方法就很方便了。

如果对连接表和多对多关联之类的概念还不熟悉，那么花些时间看看数据建模的很好的介绍是值得的。对这些概念的了解也有助于数据驱动的项目的设计、理解以及交流。George Reese的《Java Database Best Practices》（O'Reilly）就有这样的介绍，而且还可以在线阅读这本书的部分章节，网址是 <http://www.oreilly.com/catalog/javadtbp/chapter/ch02.pdf>。

由映射文档提供的调整和配置选项（尤其是使用`meta`标签时），为如何构建源代码和数据库模式带来了很大的灵活性。你亲自编写这些配置文件所获得的控制能力，没什么可以与之相比，但是，对于大多数需要和应用场合来说，使用映射驱动（`mapping-driven`）的生成工具就已经足够了。这些生成工具的一个最大优点就是，它们可以免去你输入繁琐的文件内容！创建好了`Artist.hbm.xml`文件以后，我们需要将它添加到`hibernate.cfg.xml`的映射资源部分。在`src`目录中打开`hibernate.cfg.xml`文件，添加例4-2中用粗体字显示的那一行。

例4-2：将`Artist.hbm.xml`添加到Hibernate配置文件中

```
<?xml version='1.0'encoding='utf-8'?>
<! DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
```

```
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
.....
<mapping resource="com/oreilly/hh/data/Track.hbm.xml"/>
<mapping resource="com/oreilly/hh/data/Artist.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

准备好后，我们也为Track类增加一个Artists集合。编辑Track.hbm.xml文件，添加一个新的artists属性，如例4-3所示（新添加的内容以粗体显示）。

例4-3：在Track映射文件中增加一个artist集合

```
.....
<property name="playTime" type="time">
<meta attribute="field-description">Playing time</meta>
</property>
<set name="artists" table="TRACK_ARTISTS">
<key column="TRACK_ID"/>
<many-to-many
class="com.oreilly.hh.data.Artist" column="ARTIST_ID"/>
</set>
<property name="added" type="date">
<meta attribute="field-description">When the track was created
</meta>
</property>
.....
```

这样会新增加一个名为artists的Set类型的属性。它同样使用例4-1提到的TRACK_ARTISTS连接数据表来链接到我们所映射的Artist对象。这种双向关联（bidirectional association）非常有用。其中，需要将关联的一端标记成“反向”（inverse），让Hibernate明确知道究竟在做什

么，这一点很重要。在这种多对多关联的情况下，虽然inverse属性可以影响Hibernate在什么时候自动更新连接表，不过，选择哪一端作为反转映射并不重要。如果只是从试图理解数据库的人的角度来考虑，既然将连接数据表命名为"TRACK_ARTISTS"，也就表明从艺人（ARTISTS表）链接回曲目（TRACK表）是反向端的最佳选择。

Hibernate本身并不在意我们选择哪一端，只要我们将其中一端标识为inverse就可以。我们在例4-1中就是这样配置的。在这样的配置下，如果我们对Track对象内的artists集合做出了修改，Hibernate将知道它需要更新TRACK_ARTISTS表。而如果我们对Artist对象中的tracks集合做出了修改，Hibernate不会自动更新。

更新Track映射文档时，我们也可以像补充Artist的name属性那样，来充实title属性的配置：

```
.....
<property name="title" type="string">
  <meta attribute="use-in-tostring">true</meta>
  <column name="TITLE" not-null="true" index="TRACK_TITLE"/>
</property>
.....
```

有了这个更新过的映射文件，我们可以再次重新执行ant codegen来更新Track的源代码，并创建新的Artist类的源代码。如果你查看Track.java文件，会看到已经新增加了一个类型为Set的artists属性，还新增加了一个toString（）方法。例4-4是新的Arist.java的内容。

例4-4: 为Artist类生成的代码

```
package com.oreilly.hh.data;
//Generated Sep 3, 2007 10: 12: 45 PM by Hibernate Tools 3.2.0.b9
import java.util.HashSet;
import java.util.Set;
/**
 *Represents an artist who is associated with a track or album.
 *@author Jim Elliott (with help from Hibernate)
 */
public class Artist implements java.io.Serializable{
    private int id;
    private String name;
    /**
     *Tracks by this artist
     */
    private Set tracks=new HashSet (0) ;
    public Artist () {
    }
    public Artist (String name) {
        this.name=name;
    }
    public Artist (String name, Set tracks) {
        this.name=name;
        this.tracks=tracks;
    }
    public int getId () {
        return this.id;
    }
    protected void setId (int id) {
        this.id=id;
    }
    public String getName () {
        return this.name;
    }
    public void setName (String name) {
        this.name=name;
    }
    /**
     **Tracks by this artist
     */
    public Set getTracks () {
        return this.tracks;
    }
    public void setTracks (Set tracks) {
        this.tracks=tracks;
    }
}
```

```

}
/**
 *toString
 *@return String
 */
public String toString () {
    StringBuffer buffer=new StringBuffer () ;
    buffer.append (getClass () .getName () ) .append ("@" ) .append (
    Integer.toHexString (hashCode () ) ) .append ("[" ) ;
    buffer.append ("name" ) .append ("=" ) .append (getName () ) .append
    ("'" ) ;
    buffer.append ("]" ) ;
    return buffer.toString () ;
}
}

```

注意：有人在给Hibernate挑毛病吗？修改一下代码生成工具，在toString () 方法中使用StringBuilder，而不是StringBuffer，怎么样？

为什么不能正常运行

如果你看到Hibernate报告以下几行信息，先不要失望：

```

[hibernatetool]An exception occurred while running exporter
#2: hbm2java (Generates a set of.java files)
[hibernatetool]To get the full stack trace run ant with-verbose
[hibernatetool]org.hibernate.MappingNotFoundException: resource:
com/oreilly/hh/data/Track.hbm.xml not found
[hibernatetool]A resource located at
com/oreilly/hh/data/Track.hbm.xml was not
found.
[hibernatetool]Check the following:
[hibernatetool]
[hibernatetool]1) Is the spelling/casing correct?
[hibernatetool]2) Is com/oreilly/hh/data/Track.hbm.xml available
via the c
lasspath?
[hibernatetool]3) Does it actually exist?
BUILD FAILED

```

这只是因为从新下载的示例目录中运行程序，**Ant**还没有准备好类路径（**classpath**）声明，我们以前在第2章的2.3节讨论过这个问题。如果你再试着运行一次（或者在第一次运行之前，先手工运行一次**ant prepare**命令），就没有问题了。

其他

考虑过现在（**Java 5**）提倡的类型安全（**type-safety**）的问题吗？目前生成的这些类使用的都是**Collections**类的非泛型化（**nongeneric**）版本，用当前新版本的**Java**编译器来编译这些代码时，编译器会报告类似以下的信息：

```
[javac]Note: /Users/jim/svn/oreilly/hib_dev_2e/current/scratch/ch
04
/src/com/oreilly/hh/CreateTest.java uses unchecked or unsafe
operations.
[javac]Note: Recompile with-Xlint: unchecked for details.
```

如果有办法可以生成使用**Java**泛型（**Generics**）的代码，那结果一定非常棒，这样就可以加强对放入**tracks**集合（**Set**）的东西的控制，同时也避免了这些编译警告；也不用像原来使用**Collections**时，需要进行繁琐的类型转换。还好，非常幸运，只要修改一下我们调用**hbm2java**的方式，问题就可以解决了。编辑**build.xml**文件，将**hmb2java**一行修改为以下样子：

```
<hbm2java jdk5="true"/>
```

这是在告诉生成工具，我们正在使用Java 5（或更高版本）环境，这样就可以利用新JDK提供的有用的新功能了。经过这一修改后，再次运行ant codegen，注意观察一下新生成的代码中的变化，如例4-5中突出显示的部分所示。

例4-5：用jdk5方式生成Artist类的改进

```
.....
private Set<Track> tracks=new HashSet<Track> (0) ;
.....
public Artist (String name, Set<Track> tracks) {
    this.name=name;
    this.tracks=tracks;
}
.....
public Set<Track> getTracks () {
    return this.tracks;
}
public void setTracks (Set<Track> tracks) {
    this.tracks=tracks;
}
.....
```

这就是我希望看到的代码，使用Java 5的Collections新增的泛型功能，漂亮地实现了类型安全的集合。对Track.java中的artists属性也进行类似的处理。这里先以新的Track类“完整”（full）构造函数作为例子，稍后将在例4-9中看到如何调用该构造函数：

```
public Track (String title, String filePath, Date playTime,
    Set<Artist> artists, Date added, short volume) {
.....
```


}

注意：噢，这样就好多了。这么个不起眼的配置参数，竟带来这么大的好处。

在这个新的构造函数中，为playTime和Data added属性之间的artists属性创建了一个参数化的Set类型的参数。

现在已经创建（或更新）好了类，我们就可以使用ant schema来建立支持它们的新数据库模式了。

当然，在生成源代码和建立数据库模式时应该注意有没有出现错误消息，以免映射文档中有任何语法或概念性错误。不过，并非所有出现的异常都是你必须解决的实际问题。我在试验如何改进这个数据库模式时，遇到了几个异常，报告Hibernate试图删除以前运行时并没有建立过的外键（foreign key）约束。模式生成工具并不予以理会，而继续进行下去。这看起来很可怕，但却可以正常运行。这一点在未来版本中可能会有所改进（Hibernate或HSQLDB，或者也许只是修改SQL方言的实现），不然，也要学着忍受这些小缺点，它们已经存在好几年了。

生成的数据库模式中包含我们期待的数据库表，其中有一些索引和外键约束的设置。随着我们的对象模型越来越复杂，Hibernate所做

的工作量（以及专业性难度）也会逐渐增加。模式生成工具的完整输出相当冗长，所以例4-6仅列出一些重点内容。

例4-6：摘录自新生成的数据库模式的部分内容

```
[hibernatetool]drop table ARTIST if exists;
[hibernatetool]drop table TRACK if exists;
[hibernatetool]drop table TRACK_ARTISTS if exists;
[hibernatetool]create table ARTIST (ARTIST_ID integer generated
by default
as identity (start with 1) , NAME varchar (255) not null,
primary key (ARTIST_ID) , unique (NAME) ) ;
[hibernatetool]create table TRACK (TRACK_ID integer generated by
default as
identity (start with 1) , TITLE varchar (255) not null,
filePath varchar (255) not null, playTime time, added date,
volume smallint not null, primary key (TRACK_ID) ) ;
[hibernatetool]create table TRACK_ARTISTS (ARTIST_ID integer not
null,
TRACK_ID integer not null, primary key (TRACK_ID, ARTIST_ID) ) ;
[hibernatetool]create index ARTIST_NAME on ARTIST (NAME) ;
[hibernatetool]create index TRACK_TITLE on TRACK (TITLE) ;
[hibernatetool]alter table TRACK_ARTISTS add constraint
FK72EFDAD8620962DF foreign key (ARTIST_ID) references ARTIST;
[hibernatetool]alter table TRACK_ARTISTS add constraint
FK72EFDAD82DCBFAB5 foreign key (TRACK_ID) references TRACK;
```

注意：酷！我甚至还不知道如何在HSQLDB中做这些事！

图4-1是新增加了这些内容后，数据库模式在HSQLDB中的树形视图。我不确信为什么为艺人的姓名字段（NAME字段）要用两个单独的索引来建立惟一性约束限制，但这似乎是HSQLDB本身特殊的实现方法，不过这种方式运行得很好。

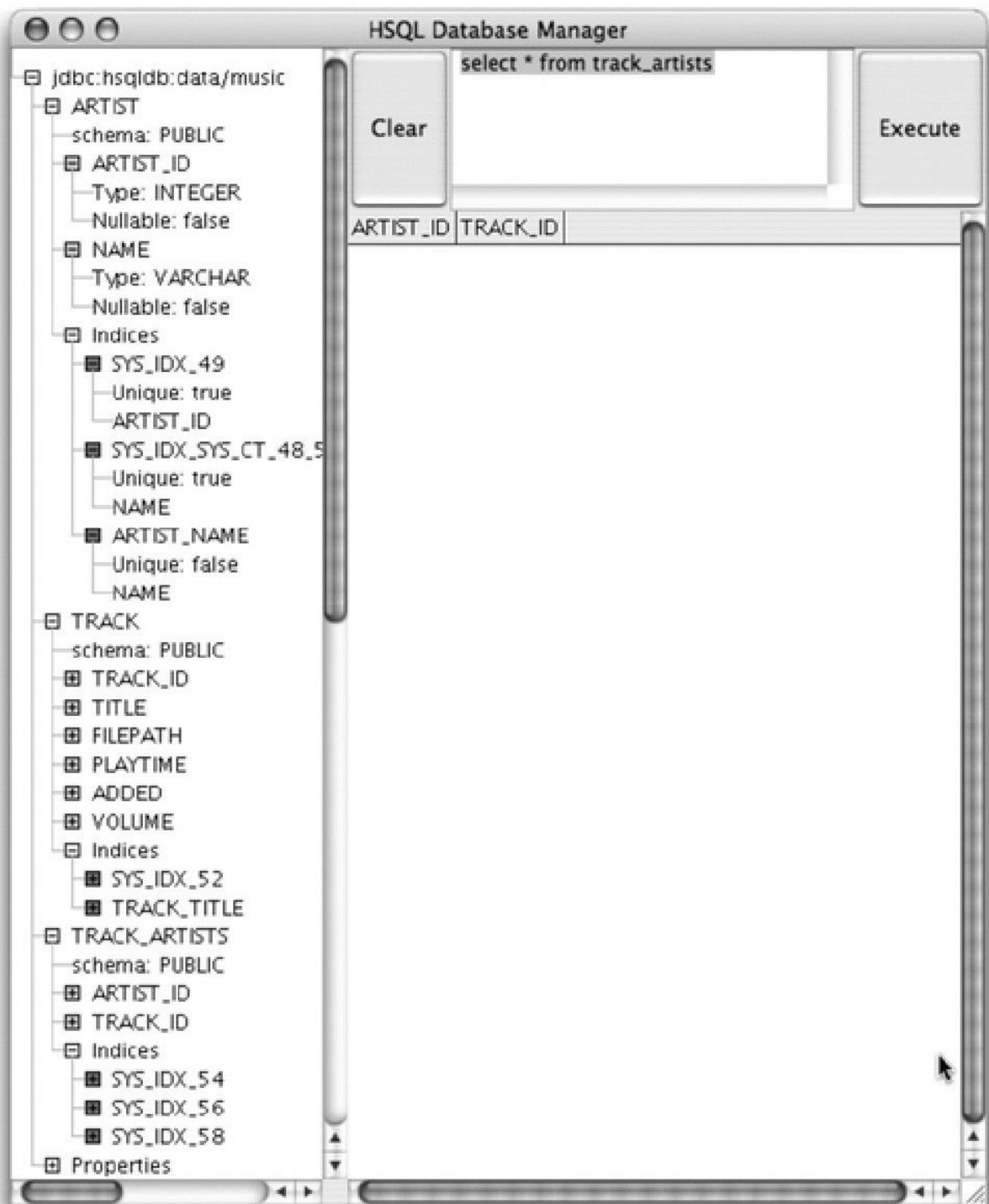


图 4-1 更新数据库模式后的HSQLDB树状图形界面

发生了什么事

我们已经建立了一个对象模型，让Track和Artist对象可以记录彼此之间的任何一组关系。任何曲目都可以关联到任意数目的艺人，而任何艺人也可以关联任意数目的曲目。要将这种模型正确地建立起来可能确实不容易，尤其是对于面向对象编程或关系数据库的新手（或者二者都是新手！），所以，有Hibernate的帮助真的很好。但是等一下，等你看到用这样建立起来的模型来处理数据会有多么方便时，你更会确认这一点。

值得强调的是，艺人和曲目之间的连接关系并没有保存在ARTIST表或TRACK表自身内。因为二者之间是多对多关联，这意味着一位艺人可以关联到多个曲目，而一个曲目也可以关联到多位艺人，这些连接关系都保存在另一个名为TRACK_ARTISTS的单独的连接表中。这张数据库表中的每一条记录都有一对ARTIST_ID和TRACK_ID，用于代表特定艺人关联到特定曲目。通过创建和删除这张表中的记录行，我们就可以建立任何需要的关联模式。（这就是为什么多对多关系总是要用关系数据库来表达的原因所在。前面提到的《Java Database Best Practices》一书中，George Reese对这种数据模型有很好的介绍。）

记住这一点以后，你也会注意到生成的类不包含任何用于管理TRACK_ARTISTS表的代码，而后面用于创建和链接持久化Track和

Artist对象的示例也没有。不需要这样的操作，是因为Hibernate中特殊的Collection类会利用例4-1和例4-3中增加的映射信息来为我们处理好这些细节。好了，来创建一些曲目和艺人对象吧。

集合的持久化

我们的第一个任务就是改进CreateTest类：利用数据库模式中新增的内容，创建一些艺人对象，并为它们关联上一些曲目对象。

应该怎么做

首先，在CreateTest.java中增加一些辅助方法，以简化我们的任务，如例4-7所示（修改和新增内容以粗体显示）。

例4-7：用于查找和创建艺人对象，并将它们链接到曲目对象的工具方法

```
package com.oreilly.hh;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import com.oreilly.hh.data.*;
import java.sql.Time;
import java.util.*; ❶
/**
 *Create more sample data, letting Hibernate persist it for us.
 */
public class CreateTest{
/**
 *Look up an artist record given a name.
 *@param name the name of the artist desired.
 *@param create controls whether a new record should be created if
 *the specified artist is not yet in the database.
 *@param session the Hibernate session that can retrieve data
 *@return the artist with the specified name, or<code>null</code>
>if no
 *such artist exists and<code>create</code>is<code>>false
</code>.
```

```

    *@throws HibernateException if there is a problem.
    */
    public static Artist getArtist (String name, boolean create,
Session session) ❷
    {
        Query query=session.getNamedQuery
("com.oreilly.hh.artistByName");
        query.setString ("name", name);
        Artist found= (Artist) query.uniqueResult (); ❸
        if (found==null&&create) {❹
            found=new Artist (name, new HashSet<Track> () );
            session.save (found);
        }
        return found;
    }
    /**
    *Utility method to associate an artist with a track
    */
    private static void addTrackArtist (Track track, Artist artist)
{❺
    track.getArtists ().add (artist);
}

```

和处理Hibernate经常使用的方法一样，这段代码相当简单，一目了然：

❶我们前面导入过`java.util.Date`，但现在需要导入整个`util`包，才能使用`Collection`接口。粗体的“*”就是为了突出这一点，不过浏览例子时容易忽略它。

❷如果我们为同样的艺人创建多个曲目的话，则希望可以重用同样的数据（这就是使用`Artist`对象，而不只是存储字符串的全部原因）。`getArtist ()`方法完成按名字查找艺人的功能。

❸ `uniqueResult()` 方法是 `Query` 接口的一个方便特色，尤其适合于以下情况：我们知道查询要么只有一个结果，要么没有任何结果。这样就免去了获取结果列表、检查列表长度。如果包含数据的话，再提取第一个结果元素，这么多繁琐的步骤。使用这个方法要么只取回一个结果；或者当没有结果时，就返回 `null`（如果查询返回多个结果，这个方法将抛出一个异常。你可能会认为我们在列上施加的 `unique` 约束能够防止这种异常，但 `SQL` 是大小写敏感的，而我们的查询匹配是大小写不敏感的。所以在创建一个新记录之前，应该总是先调用 `getArtist()`，以确保同名的艺人是否已经存在）。

❹ 所以我们需要做的就是检查 `null` 值，如果没有找到相应名字的艺人，而且 `create` 标志也表明我们想要创建艺人对象时，就创建一个新的 `Artist`。

如果我们省去 `session.save()` 调用，那么所有艺人 `Artist` 对象将保持瞬时状态。`Hibernate` 这时也很有帮助，如果我们试图在这种情况下提交事务，`Hibernate` 就会检查到持久化 `Track` 实例引用了瞬时状态的 `Artist` 实例，从而抛出一个异常。你可以回顾一下第3章对生命周期的讨论，以及第5章的“生命周期关联”，它们更深入地探究了这一问题。

❺ `addTrackArtist()` 方法差不多简单得令人尴尬。它只是普通的 `Java Collection` 代码，获取属于某个 `Track` 的艺人对象集合 (`Set`)，再将指定的 `Artist` 添加到这个集合中。这样真可以实现我们需要的所有功能

吗？我们通常不得不写的数据库处理代码都跑到哪儿去了？欢迎来到对象/关系映射工具的精彩世界！

可以看到`getArtist ()`方法内部使用一个命名查询来检索`Artist`记录。我们在`Artist.hbm.xml`的末尾添加了这个命名查询的定义，如例4-8所示。（实际上，可以将命名查询放在任意映射文件中，但这是最合适的地方，因为这个查询和`Artist`记录相关。）

例4-8：在`Artist`映射文档中添加检索查询语句

```
<query name="com.oreilly.hh.artistByName">
<![CDATA[
from Artist as artist
where upper (artist.name)=upper (: name)
]]>
</query>
```

该命名查询使用`upper ()`函数对艺人的姓名进行大小写不敏感（`case-insensitive`）的比较，这样，即使查询时所用的大小写与数据库中保存的数据不一样，也可以检索到相应的艺人。这种不区分大小写，但又能保留其原有大小写的方法是一种用户友好的方法，用户都喜欢这种实现方式，所以值得我们尽可能实现。除了`HSQldb`，其他数据库中将字符串转换成大写的函数可能有不同的名称，但应该都有。我们将在第8章介绍一种面向`Java`的、独立于数据库的方法，以一种漂亮的方式来实现这种字符串转换。

现在，我们以此为基础来真正创建一些链接了艺人的曲目对象。

例4-9展示了CreateTest类的剩余部分，新增部分以粗体字显示。按照示例所演示的，编辑你的源文件（或直接下载以节省打字时间）。

例4-9: 修改CreateTest.java的main () 方法，增加艺人关联数据

```
public static void main (String args[]) throws Exception{
//Create a configuration based on the XML file we've put
//in the standard place.
Configuration config=new Configuration () ;
config.configure () ;
//Get the session factory we can use for persistence
SessionFactory sessionFactory=config.buildSessionFactory () ;
//Ask for a session using the JDBC information we've configured
Session session=sessionFactory.openSession () ;
Transaction tx=null;
try{
//Create some data and persist it
tx=session.beginTransaction () ;
Track track=new Track ("Russian Trance",
"vol2/album610/track02.mp3",
Time.valueOf ("00: 03: 30") ,
new HashSet<Artist> () , ❶
new Date () , (short) 0) ;
addTrackArtist (track, getArtist ("PPK", true, session) ) ;
session.save (track) ;
track=new Track ("Video Killed the Radio Star",
"vol2/album611/track12.mp3",
Time.valueOf ("00: 03: 49") , new HashSet<Artist> () ,
new Date () , (short) 0) ;
addTrackArtist (track, getArtist ("The Buggles", true,
session) ) ;
session.save (track) ;
track=new Track ("Gravity's Angel",
"vol2/album175/track03.mp3",
Time.valueOf ("00: 06: 06") , new HashSet<Artist> () ,
new Date () , (short) 0) ;
addTrackArtist (track, getArtist ("Laurie Anderson", true,
session) ) ;
session.save (track) ;
track=new Track ("Adagio for Strings (Ferry Corsten Remix) ", ❷
"vol2/album972/track01.mp3",
```

```

        Time.valueOf ("00: 06: 35") , new HashSet<Artist> () ,
        new Date () , (short) 0) ;
        addTrackArtist (track, getArtist ("William Orbit", true,
session) ) ;
        addTrackArtist (track, getArtist ("Ferry Corsten", true,
session) ) ;
        addTrackArtist (track, getArtist ("Samuel Barber", true,
session) ) ;
        session.save (track) ;
        track=new Track ("Adagio for Strings (ATB Remix) ",
"vol2/album972/track02.mp3",
Time.valueOf ("00: 07: 39") , new HashSet<Artist> () ,
new Date () , (short) 0) ;
        addTrackArtist (track, getArtist ("William Orbit", true,
session) ) ;
        addTrackArtist (track, getArtist ("ATB", true, session) ) ;
        addTrackArtist (track, getArtist ("Samuel Barber", true,
session) ) ;
        session.save (track) ;
        track=new Track ("The World'99",
"vol2/singles/pvw99.mp3",
Time.valueOf ("00: 07: 05") , new HashSet<Artist> () ,
new Date () , (short) 0) ;
        addTrackArtist (track, getArtist ("Pulp Victim", true,
session) ) ;
        addTrackArtist (track, getArtist ("Ferry Corsten", true,
session) ) ;
        session.save (track) ;
        track=new Track ("Test Tone 1", ③
"vol2/singles/test01.mp3",
Time.valueOf ("00: 00: 10") , new HashSet<Artist> () ,
new Date () , (short) 0) ;
        session.save (track) ;
        //We're done; make our changes permanent
        tx.commit () ;
    }catch (Exception e) {
        if (tx!=null) {
            //Something went wrong; discard all partial changes
            tx.rollback () ;
        }
        throw new Exception ("Transaction failed", e) ;
    }finally{
        //No matter what, close the session
        session.close () ;
    }
    //Clean up after ourselves
    sessionFactory.close () ;
}

```

}

例4-9对现有程序代码的修改相当有限：

❶前面几行代码用于创建第3章中的3个曲目对象，这里只需要为每个曲目对象提供一个新的参数来作为Artist关联的最初空集合。随后每个再用一行代码来为曲目建立艺人的关联。我们原本可以用不同的结构来实现这段代码，编写一个工具方法以创建包含艺人对象的最初的HashSet，这样就能用一行代码完成所有的处理。不过，对于多艺人的曲目对象，我们实际使用的这种方法的适应性更好，下一节将演示这种情况。

❷最大一段新代码是简单地添加了3个新的曲目，以演示如何处理每个曲目有多个关联艺人的情况。如果你喜欢电子音乐和舞曲（或者古典音乐之类的曲目），就应该知道这个问题多么重要。因为我们将链接表达为集合对象，所以维护关联就简化为将每个艺人对象添加到相应的曲目对象就可以了。

❸最后，我们添加了一个没有艺人关联的曲目对象，看看会发生什么。现在，你可以运行`ant ctest`来创建新的样例数据，其中包含了曲目、艺人以及他们之间的关联。

如果需要对测试数据创建程序进行修改，又想再次从空数据库开始运行程序，一个有用的技巧是执行`ant schema ctest`命令。这个命令是

告诉Ant先后分别执行schema和ctest构建目标。执行schema构建目标会将现有数据全部清空，接着再执行ctest以重建数据。

注意：当然，现实生活中将数据放到数据库会采用其他做法——通过用户界面或将实际的曲目数据直接导入数据库。不过，如果只是单元测试，代码看起来就是这个样子。

发生了什么事

运行ctest后，除了Hibernate使用的SQL语句（如果仍然将hibernate.cfg.xml中的show_sql设置为true），没有什么非常有意义的输出。可以打开data/music.script来看一看里面新创建了什么，或者用ant db命令打开数据库管理器图形界面来查看。看看这三个数据库表的内容。图4-2显示了连接表中代表艺人和曲目之间关联的结果。原始数据已经隐藏起来了。如果你习惯使用关系数据库模型，则这样的查询结果表示一切都运行正常；如果你和我一样都是凡人，那么下一节介绍的内容将更加令人信服，当然也更加有趣。

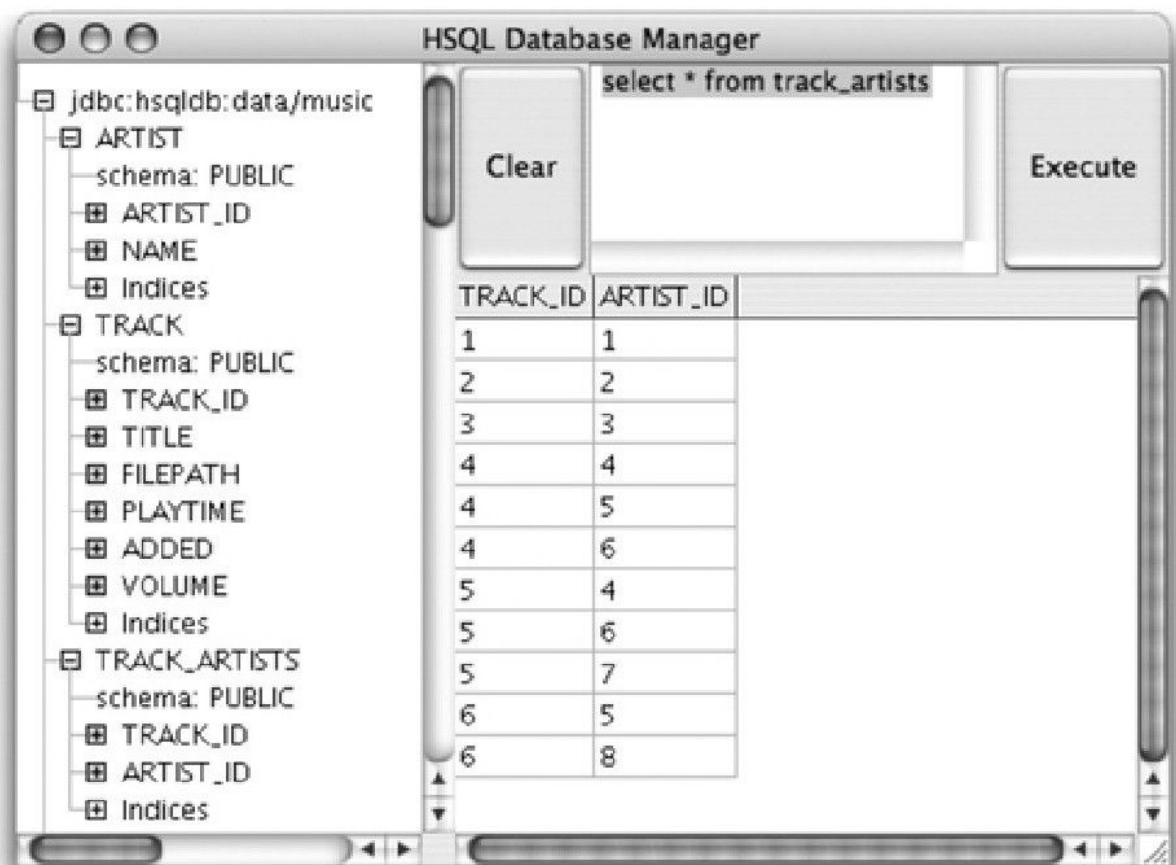


图 4-2 新版CreateTest创建的艺人和曲目之间的关联

集合的检索

你可能会认为从数据库中把集合信息检索出来同样也是这么简单。没错！这一节将改进一下我们的QueryTest类，把艺人及相关的曲目显示出来。例4-10以粗体字展示了适当的修改和增加的代码。而且只需要新增少量的代码。

例4-10: 改进的QueryTest.java (以便显示关联了曲目的艺人对象)

```
package com.oreilly.hh;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import com.oreilly.hh.data.*;
import java.sql.Time;
import java.util.*;
/**
 *Retrieve data as objects
 */
public class QueryTest{
/**
 *Retrieve any tracks that fit in the specified amount of time.
 *
 *@param length the maximum playing time for tracks to be
returned.
 *@param session the Hibernate session that can retrieve data.
 *@return a list of{@link Track}s meeting the length restriction.
 */
public static List tracksNoLongerThan (Time length, Session
session) {
    Query query=session.getNamedQuery (
        "com.oreilly.hh.tracksNoLongerThan" );
    query.setTime ("length", length) ;
    return query.list () ;
}
```

```

/**
 *Build a parenthetical, comma-separated list of artist names.
 *@param artists the artists whose names are to be displayed.
 *@return the formatted list, or an empty string if the set was
empty.
 */
public static String listArtistNames (Set<Artist>artists) {❶
    StringBuilder result=new StringBuilder () ;
    for (Artist artist: artists) {
        result.append ( (result.length () ==0) ?" (: ", " ) ;
        result.append (artist.getName () ) ;
    }
    if (result.length () >0) {
        result.append (" " ) ;
    }
    return result.toString () ;
}
/**
 *Look up and print some tracks when invoked from the command
line.
 */
public static void main (String args[]) throws Exception{
    //Create a configuration based on the XML file we've put
    //in the standard place.
    Configuration config=new Configuration () ;
    config.configure () ;
    //Get the session factory we can use for persistence
    SessionFactory sessionFactory=config.buildSessionFactory () ;
    //Ask for a session using the JDBC information we've configured
    Session session=sessionFactory.openSession () ;
    try{
        //Print the tracks that will fit in seven minutes
        List tracks=tracksNoLongerThan (Time.valueOf ("00: 07: 00") , ❷
        session) ;
        for (ListIterator iter=tracks.listIterator () ;
        iter.hasNext () ; ) {
            Track aTrack= (Track) iter.next () ;
            System.out.println ("Track: \""+aTrack.getTitle () +"\""+
            listArtistNames (aTrack.getArtists () ) +❸
            aTrack.getPlayTime () ) ;
        }
    }finally{
        //No matter what, close the session
        session.close () ;
    }
    //Clean up after ourselves
    sessionFactory.close () ;
}

```



```
}
```

❶我们增加的第一个方法是一个小工具方法，用于漂亮地格式化一组艺人的名字，将艺人名字放在一对圆括号内，并用逗号“，”作为分隔符，再加上适当的空格。或者如果艺人集合为空的话，就什么也不显示。

❷对于新加的所有有趣的多艺人相关的曲目，其播放时间都超过5分钟，所以我们将查询限值增加到7分钟，这样才能看到结果。

❸最后，我们在`println()`语句的适当位置上调用`listArtistNames()`方法，描述找到的曲目信息。现在先去掉Hibernate的查询调试输出配置，因为这些调试信息会妨碍我们观察真正想要看到的信息。编辑src目录下的`hibernate.cfg.xml`文件，将`show_sql`属性修改为`false`。

```
.....  
<!--Echo all executed SQL to stdout-->  
<property name="show_sql">false</property>  
.....
```

这样设置以后，例4-11演示了执行`ant qtest`命令后，新的输出结果。

例4-11: QueryTest输出的艺人信息

```
%ant qtest  
Buildfile: build.xml  
prepare:
```

```
compile:
[javac]Compiling 1 source file
to/Users/jim/svn/oreilly/hib_dev_2e
/current/scratch/ch04/classes
qtest:
[java]Track: "Russian Trance" (PPK) 00: 03: 30
[java]Track: "Video Killed the Radio Star" (The Buggles) 00: 03:
49
[java]Track: "Gravity's Angel" (Laurie Anderson) 00: 06: 06
[java]Track: "Adagio for Strings (Ferry Corsten Remix) " (Ferry
Corsten,
William Orbit, Samuel Barber) 00: 06: 35
[java]Track: "Test Tone 1"00: 00: 10
BUILD SUCCESSFUL
Total time: 2 seconds
```

你会注意到两件事。首先，可以看到这种结果显示方式比图4-2中简单的数字列表更容易解读。其次，运行无误！即使不带任何艺人映射数据、纯测试用的“特殊”曲目都没有问题，因为Hibernate采用的方法比较友好，在这种特殊情况下会创建一个不含艺人对象的空Set对象。免去了我们为防止引发NullPointerException（空指针错误）异常，而自己编写代码检测对象是否为null的需要。

注意：但是，等一下，还有呢！不需要编写额外的代码.....

使用双向关联

在新建数据的代码中，我们简单地通过把Java对象添加到适当的集合，就建立起曲目对象到艺人对象的链接。Hibernate会把这些关联和对象分组转化成它为此所创建的连接表中必要的隐含项。这样，我们就能用简单易读的代码来建立和维护这些关联关系。但是要记住，我们在这里创建的这个关联是双向的，Artist类中也有一个内含一些Track对象集合的关联。不用烦恼应该在那存储什么。

这种设计的方便性在于，只有当对“主要映射”（primary mapping）做出修改时，才会对修改传播到反向映射端。如果只对反向映射端做出修改（就此例而言，就是修改Artist对象中保存曲目的Set集合对象），那么修改结果不会保存下来。可惜，这就要求你在代码中要注意哪个映射是反向映射端。

好消息是我们不用为此而必须做些什么。因为我们在Artist映射文档中将这个关联标识为反向映射（inverse="true"），Hibernate知道当为一个Track对象新增一个Artist关联时，另一层没说的话就是也把该Track作为关联而添加给了那个Artist。

我们来构建一个简单的交互式图形应用程序，以帮助检查艺人对曲目的关联链接是否正确。这个应用程序可以让你输入艺人的姓名，

然后显出与该艺人相关联的曲目。大部分代码和第一个查询测试程序非常类似。现在，创建QueryTest2.java文件，并输入例4-12所示的程序代码。

例4-12: QueryTest2.java的源代码

```
package com.oreilly.hh;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import com.oreilly.hh.data.*;
import java.sql.Time;
import java.util.*; import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/**
 *Provide a user interface to enter artist names and see their
tracks.
 */
public class QueryTest2 extends JPanel{
    JList list; //Will contain tracks associated with current artist
    DefaultListModel model; //Lets us manipulate the list contents
    /**
    *Build the panel containing UI elements
    */
    public QueryTest2 () {
        setLayout (new BorderLayout () ) ;
        model=new DefaultListModel () ;
        list=new JList (model) ;
        add (new JScrollPane (list) , BorderLayout.SOUTH) ;
        final JTextField artistField=new JTextField (28) ;
        artistField.addKeyListener (new KeyAdapter () {❶
            public void keyTyped (KeyEvent e) {❷
                SwingUtilities.invokeLater (new Runnable () {❸
                    public void run () {
                        updateTracks (artistField.getText () ) ;
                    }
                }) ;
            }
        }) ;
        add (artistField, BorderLayout.EAST) ;
        add (new JLabel ("Artist: ") , BorderLayout.WEST) ;
    }
}
```

```

/**
 *Update the list to contain the tracks associated with an artist
 */
private void updateTracks (String name) {④
model.removeAllElements () ; //Clear out previous tracks
if (name.length () <1) return; //Nothing to do
try{
//Ask for a session using the JDBC information we've configured
Session session=sessionFactory.openSession () ; ⑤
try{
Artist artist=CreateTest.getArtist (name, false, session) ;
if (artist==null) { //Unknown artist
model.addElement ("Artist not found") ;
return;
}
//List the tracks associated with the artist
for (Track aTrack: artist.getTracks () ) {⑥
model.addElement ("Track: \""+aTrack.getTitle () +
 "\", "+aTrack.getPlayTime () ) ;
}
}finally{⑦
//No matter what, close the session
session.close () ;
}
}catch (Exception e) {
System.err.println ("Problem updating tracks: "+e) ;
e.printStackTrace () ;
}
}

private static SessionFactory sessionFactory; //Used to talk to
Hibernate
/**
 *Set up Hibernate, then build and display the user interface.
 */
public static void main (String args[]) throws Exception{
//Load configuration properties, read mappings for persistent
classes
Configuration config=new Configuration () ; ⑧
config.configure () ;
//Get the session factory we can use for persistence
sessionFactory=config.buildSessionFactory () ;
//Set up the UI
JFrame frame=new JFrame ("Artist Track Lookup") ; ⑨
frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE) ;
frame.setContentPane (new QueryTest2 () ) ;
frame.setSize (400, 180) ;
frame.setVisible (true) ;
}

```

}

注意：没错，这是露骨的推销。

示例中有一大段新增加的代码用于建立一个Swing用户界面，这实际上还是个相当原始的界面，无法漂亮地调整窗口大小。但是，如果要处理这些界面效果上的细节，程序会变得更长，其实这些内容也不是本书应该讨论的范围。如果你想看一些如何建立丰富、高质量的Swing界面的例子，可以参考《Java Swing》（O'Reilly, Second Edition），这是一本很厚的巨著，介绍了Swing界面开发的方方面面。

❶在构造函数中我想重点介绍的只有KeyListener，要将它加到artistField。这段代码相当有技巧，它创建了一个匿名（anonymou）类，无论用户何时在艺人文本框中输入信息，都会调用这个匿名类的keyTyped（）方法。

❷这个方法检查输入文本框当前是否包含一个可识别的艺人名字，并更新相应的曲目显示。

❸不幸的是，在调用这个方法时，还没有更新文本框以反映最新的键盘输入，所以我们不得不通过SwingUtilities的invokeLater（）方法将实际的显示更新推迟到另一个匿名类（Runnable的实例）中。这种技巧可以在Swing“有时间处理来处理”时，再进行更新。在我们这个例子中意味着文本输入框将自己完成更新。

④在用户输入艺人名字时调用的`updateTracks ()`方法正是有趣的Hibernate处理发生的地方。首先，这个方法清理干净列表，也就是清除之前显示过的任何曲目。如果艺人名字为空，就做到这里为止。

⑤否则，这个方法会打开一个Hibernate会话，试着用我们在`CreateTest`里所写的`getArtist ()`方法查找艺人信息。这一次我们告诉这个方法，如果找不到相应的艺人，就不要创建艺人对象，所以如果用户没有输入已知艺人的名字，就会得到一个`null`。

⑥另一方面，如果确实找到一个`Artist`记录，则遍历该艺人的所有关联曲目集合中找到的`Track`记录，并显示每个曲目的相关信息。这将测试逆向（`inverse`）关联是否按照我们的预想来工作。

⑦最后（这里不是故意用双关语），要确保退出该方法时关闭了会话，即使是因为异常而退出。你肯定不会希望发生会话泄露，不过，如果你想搞垮整个数据库环境的话，这就是个好方法。

⑧`main ()`方法的开始还是相同的Hibernate配置步骤，我们以前就看过了。

⑨接着，会创建并显示用户界面框架，再设置当关闭用户界面时就结束程序。显示这个界面框架后，`main ()`返回，从那时起，就由Swing事件循环控制后续流程。

在创建（或下载）好代码文件以后，还需要在build.xml（Ant构建文件）的末尾新增加一个构建目标（如例4-13所示），才能够调用这个新创建的类。

注意：这非常类似于现有的qtest构建目标，直接复制、调整一下就可以了。

例4-13：运行新的查询测试程序的Ant构建目标

```
<target name="qtest2" description="Run a simple Artist
exploration GUI"
  depends="compile">
  <java classname="com.oreilly.hh.QueryTest2" fork="true">
  <classpath refid="project.class.path"/>
  </java>
</target>
```

现在，你可以输入ant qtest2命令来启动程序，自己体验一下这个程序。图4-3是运行中的程序界面，显示了示例数据中一位艺人的曲目数据。



图 4-3 非常简单的艺人曲目浏览器

使用简单集合

到目前为止，我们所看到的集合都含有和其他对象之间的关联数据，对于本章讨论的“集合和关联”这一主题来说，这并没有什么不妥，但这并不是Hibernate集合的惟一用法。你也可以为简单值的集合定义映射，例如字符串、数字以及非持久化的值类。

应该怎么做

假设我们想为数据库中的每个曲目都保存一定数量的评论，用一个名为comments的新属性来记录每一个相关联评论的String值。Track.hbm.xml中的新映射数据看起来很像原来我们为艺人映射文档所做的那样，只是这里更为简单（[\[1\]](#)）一些：

```
<set name="comments"table="TRACK_COMMENTS">  
  <key column="TRACK_ID"/>  
  <element column="COMMENT"type="string"/>  
</set>
```

（可以将这段配置信息放在映射文件中</class> 关闭标签之前。如果你想在Track类的构造函数中进行处理，也必须这么做。）

由于我们需要为每个Track都保存任意数量的评论，所以我们得另建一个专门用于保存评论的新数据库表。每个评论都会通过每个曲目

的id属性链接到正确的Track对象。

用ant schema命令重新构建数据库，以下显示了构建过程的内容：

```
[hibernatetool]create table TRACK_COMMENTS (TRACK_ID integer not
null, COMMENT
  varchar (255) ) ;
.....
[hibernatetool]16: 16: 55, 876 DEBUG SchemaExport: 303-alter
table TRACK_COMMENTS
  add constraint FK105B26882DCBFAB5 foreign key (TRACK_ID)
references TRACK;
```

注意：数据建模者一般将这种关联关系称为“一对多”关系。

用ant codegen更新Track类之后，还需要在CreateTest.java中每次调用构造函数时，为comments属性增加另一个Set对象。例如：

```
track=new Track ("Test Tone 1",
"vol2/singles/test01.mp3",
Time.valueOf ("00: 00: 10"), new HashSet<Artist> () ,
new Date () , (short) 0, new HashSet<String> () );
```

然后，我们可以用以下代码行来指定评论内容：

```
track.getComments () .add ("Pink noise to test equalization") ;
```

执行ant ctest可以快速编译并运行这个程序（确保不要忘记为每个曲目对象新增加第二个专门保存字符串的HashSet对象），接着再检查data/music.script，看看数据库是如何存储评论数据的。或者，在

QueryTest.java中输出曲目信息的println () 之后再多加一个循环，以打印输出刚刚显示过的曲目的评论信息：

```
for (String comment: aTrack.getComments () ) {  
    System.out.println ("Comment: "+comment);  
}
```

接着，运行ant qtest，可以得到以下的输出：

```
.....  
[java]Track: "Test Tone 1"00: 00: 10  
[java]Comment: Pink noise to test equalization
```

当工具让简单的事情变得更容易时，那真的很好。下一章我们会看到，即使是复杂的事情，也是畅通无阻。

[1] 如果要将这些示例移植到Oracle中，必须修改"COMMENT"字段的名称，因为这是Oracle SQL的一个保留字。还有很多类似的标准保留字！

第5章 更复杂的关联

当然啦，多个朋友多条路。但是这并不简单，所以应该探讨一下对象之间更丰富的关系，如何比简单的分组携带更多的信息。在本章中，我们将要探讨一下如何将曲目组合成专辑（album）。第4章没有介绍这一点，是因为组织专辑不仅仅是简单地对曲目进行分组，还需要知道曲目在专辑中的排列顺序，以及诸如它们在哪张唱片（disc）等信息，才能支持多唱片的专辑。实现这些功能光靠自动生成的连接数据库表是不够的，所以我们得自己设计AlbumTrack对象和数据表，才可以把专辑和曲目链接起来。

关联的主动加载和延迟加载

先富（rich），而后主动（eager）和延迟（lazy）？这些词听起来好像我们在把数据模型当成鲜活的人物来介绍。但这其实是O/R映射的重要主题之一。随着数据模型不断地增长，对象和数据库表之间的关联也会随之增加，程序的功能也不断增多，这很好。但是，通常最后的结果就是成堆的对象之间被链接得零零碎碎。这样，当从一大串彼此相关的对象簇中加载一个从属的对象时，会发生什么？可以看到，只要通过遍历对象的属性，就可以从一个对象访问到关联的另一个对象。这似乎是说，当需要加载某个对象时，就得必须加载相关联

的所有对象。对于小型数据库来说，这没有什么问题（事实上，我们在示例中使用的HSQLDB数据库只是在运行时，才全部存在于内存中）；但是一般情况下，应用程序预定的数据库容量肯定要超过程序自身占用的内存量。哇！但是，即使真能全部加载，也不太可能真正用到大部分数据对象，所以，加载全部对象只是浪费资源而已。

所幸，对象/关系映射软件（包括Hibernate在内）的设计者已经预见了一个问题。处理的技巧就是关联配置成"lazy"的，这样，只有在实际需要访问相关联对象的引用时，才会加载相应的对象。Hibernate将负责维护链接对象的标识，直到真正访问它时，才会加载这个对象。这是我们经常使用的集合在本质上的一个重要特性。

应该怎么做

在Hibernate 3发布以前，关联在默认情况下并不设置lazy属性，所以必须手工在映射声明中设置lazy属性。不过，因为使用延迟加载总是最佳实践，所以Hibernate 3就默认保留了lazy的设置（进行应用程序移植时需要特别小心，绝不要发生像这种向后不兼容（backward-incompatible）的问题）。

当对象的关联是lazy类型时，Hibernate就使用它自己特殊的Collections类的延迟加载实现机制，直到试图真正使用集合的内容

时，才从数据库中加载相应的内容。这些处理是完全透明的，所以代码编写者并不会注意到代码背后发生的这些细节。

嗯，如果真是这么简单就能解决加载大量相关对象的问题，那为什么还要将这个设置关掉呢？问题在于，当你关掉**Hibernate**会话时，这种透明性就消失了。在关掉会话后，如果你试图访问尚未初始化的延迟加载集合（即使将这个集合赋值给不同的变量，或者作为方法调用的返回值），**Hibernate**提供的代理集合（**proxy collection**）也将不再访问数据库以延迟加载它的内容，而是强制抛出一个**LazyInitializationException**异常（稍后我们将介绍如果确实需要很快关闭会话时，应该如何处理）。

注意：复杂度守恒看起来很像热力学定律。

因为这种无法预料的异常与**Hibernate**特定的程序代码本身没有什么关系，与加载超过实际使用数量的对象而付出的潜在代价相比，如果你认为加载所有可能需要使用的对象更为重要，那么就可以关掉**lazy**设置。你要负责仔细考虑在什么情况下需要禁用这一设置，如果使用**lazy**设置的话，也要确保安全地使用延迟加载对象。**Hibernate**参考手册中深入讨论了这方面的选择策略。

例如，如果我们不想使用延迟初始化，就可以把曲目艺人映射文件按例5-1所示的方式进行设置。

例5-1：在曲目艺人关联中使用主动加载初始化

```
<set name="artists"table="TRACK_ARTISTS"lazy="false">
  <key column="TRACK"/>
  <many-to-many
class="com.oreilly.hh.data.Artist"column="ARTIST_ID"/>
</set>
```

其他

集合以外延迟加载的用法？缓存（cache）和集群（cluster）？

延迟加载集合的支持机制很容易理解，因为Hibernate提供了它自己对Collection接口的实现。

但是，对于其他类型的关联应该如何处理？其他类型的关联也可以受益于按需加载而带来的好处。

事实上，Hibernate确实支持，用法几乎也是那么简单（至少从服务的使用者来看确实如此）。同样，从Hibernate 3开始，类的默认加载方式就是延迟加载，不过，你可以将整个持久化类设置为lazy="false"来关闭延迟加载（这个属性就放在映射文档的class标签中）。

当采用延迟加载的方式来映射一个类时，Hibernate将生成一个代理类来扩展数据类（data class）。这个延迟代理类会推迟加载数据的

时机，直到真正需要使用数据时再加载。任何关联到这个延迟加载类的其他对象都会悄悄地用这样的代理对象进行扩展，而不是引用实际的数据对象。当第一次使用代理对象的任何方法时，才会加载真正的数据对象，并将方法调用委托给数据对象。在加载完成数据对象以后，代理对象将一直把所有的方法调用都委托给它。如果想做的再花哨点，可以使用`proxy`属性来指定代理类将要扩展（或实现）的特定类（或接口）。`lazy`属性是将持久化类本身指定为要被代理的类的快捷方式。（如果看不懂，也别担心，这只是说明你现在还不需要这种功能而已。当你需要时，就会懂了！）

自然地，在会话关闭之前才能加载需要使用的任何东西，这一限制依然适用于这种延迟加载类的初始化。你可以使用延迟加载类，但这样做时，一定要小心慎重，并做好规划。

Hibernate参考文档在它的“提升性能”（Improving Performance）（[\[1\]](#)）一章中深入讨论了这些值得权衡的考虑。也介绍了Hibernate甚至可以和JVM层或集群对象缓存进行集成，以减轻数据库访问的瓶颈，提升大型分布式应用程序的性能。当集成这样的缓存机制时，可以在映射文档中用`cache`标签（足够恰当）来配置类和关联的缓存行为。这些配置方法不在本书讨论的范围内，但是你应该注意到这些可行的方法，因为说不定你的应用程序就可以从中受益。

令人头痛的问题

忽略对这些主题的讨论，可能不是你欣赏的做法。坦白地说，理解代码执行时的任何路径上需要访问的对象集的边界，在优化加载对象的同时，还不会浪费太多的内存和开发人员的精力，这些问题都是O/R映射中存在的最困难的权衡和挑战。甚至像Hibernate这样优秀的组件库也不完全屏蔽这些问题。

幸好，有些技术可以用于对数据访问代码进行优化，在许多常见的应用场合中从根本上避免整个问题，从中你可以理解这些技术得以流行的原因。请记住，只有在关闭Hibernate会话之后再去访问关联对象时，延迟加载的关联才会出问题。如果在会话打开期间可以完成所有的数据处理，那么延迟加载的关联总是可以正常工作，不必为它们过多担心。

好，那么在程序开始运行时就打开会话，直到程序结束再关闭会话，这样可以吗？哦，不，这样是不行的。因为一个会话就包括了一个数据库事务（**transaction**），而数据库是共享资源。打开的事务被保留的时间越长，数据库就越可能陷入困境；向其他用户和进程隐藏的活动越多，这些活动就越可能被发现；当你最终提交事务处理时，也就越可能与其他事务遇到冲突。所以，绝对不要在等待用户进行操作的同时而保持打开一个事务。

所以我们岂不就是遇上Catch-22（[\[2\]](#)）了？其实情况并没有想象中的那么坏。通常，只要应用程序的总体结构设计得合理，就可以让数据访问发生的位置一目了然，也就为Hibernate会话提供了开始和结束的自然边界。例如对于Web应用程序，在处理到来的请求的过程中打开Hibernate会话，这样做很有意义。事实上，开发人员经常建立一些Servlet过滤器（filter）来自动完成这些处理。此外，如果有些后台任务需要定期地处理数据（例如在夜间发送电子邮件），它们就可以在类似良好定义的边界内使用各自单独的会话。所以，虽然处理策略需要一定的技巧而且也很重要，但困难也不是不能克服的。我们将在第13章和第14章用一些具体的示例来演示一些不错的方法。

[\[1\]](#)

http://www.hibernate.org/hib_docs/v3/reference/en/html/performance.html.

[\[2\]](#) 这个词语源于美国著名作家约瑟夫·海勒的成名作《第22条军规》（Catch-22），写于1961年，这是一部典型的黑色幽默之作。在当代英语中Catch-22作为一个独立的单词，使用频率非常高，用来形容自相矛盾、不合逻辑的规定，或条件所造成的无法脱身或左右为难的困境。

有序集合

集合排序似乎与对象生命周期这一主题有些偏离，不过它确实是一个重要内容。这一章本来就打算介绍一些处理集合映射方面的有趣的技巧，所以我们就言归正传！现在我们的首要目标是存储组成专辑的曲目，并保证曲目的正确顺序。稍后，我们再加入曲目所属的唱片，以及曲目在该唱片的位置等信息，这样才可以妥善地处理包含多个唱片的专辑。

注意：喔，对，那就是我们试着要做的事.....

应该怎么做

让集合内容保持特定的顺序其实相当简单。如果我们在组织专辑的曲目时只在乎这一点，那么只要告诉Hibernate映射到一个List或Array就可以了。例5-2演示了专辑（Album）映射中我们使用的方法。

例5-2：专辑曲目的简单排序映射

```
<list name="tracks" table="ALBUM_TRACKS">
  <key column="ALBUM_ID"/>
  <list-index column="LIST_POS"/>
  <many-to-many
class="com.oreilly.hh.data.Track" column="TRACK_ID"/>
</list>
```

这与我们一直在用的set映射十分相似（虽然这里使用了一个不同的<list>标签以标明这个集合是个有序列表，因此该集合会映射到一个java.util.List类）。但是要注意，我们也必须额外增加一个list-index标签，以建立这个列表的排序字段，同时，我们也必须在数据库中新增加一个字段来保存控制曲目顺序的值。Hibernate会为我们管理这个字段的内容，用这个字段来确保将来从数据库把列表取出时，其内容会和当初存储到数据库时保持相同的顺序。这个字段的类型是整数，可能的话，可以将它作为该数据库表的组合主键（composite key）的一部分。当生成HSQLDB数据库模式定义时，就可以用例5-2的映射内容来生成例5-3所示的数据表。

例5-3：简单有序曲目列表的HSQLDB模式定义

```
[hibernatetool]create table ALBUM_TRACKS (ALBUM_ID INTEGER not null,
TRACK_ID INTEGER not null, LIST_POS INTEGER not null,
primary key (ALBUM_ID, LIST_POS))
```

理解LIST_POS字段必不可少的原因是很重要的。我们需要控制曲目在专辑中出现的顺序，但是曲目自身并没有任何属性可以让我们确定它在专辑中的顺序（想象一下，如果播放器只能以字母顺序来播放专辑的曲目时，你会多么失望——竟然不能按照曲目的艺人顺序播放）。关系数据库系统的基本特性就是，检索得到的顺序是系统认为方便的顺序，除非指定了系统应该如何对结果进行排序。LIST_POS字

段给了Hibernate一个它可以进行控制的值，可以用于确保列表总是以当初创建的顺序进行排序的。另一种思考方式是，数据项的顺序是我们想保存下来的独立信息，所以Hibernate需要有个地方来保存它。

随之而来的结果也很重要。如果数据中有些值可以提供遍历的自然顺序，那就没有必要再提供一个索引字段（`index column`），甚至也不需要使用`list`。`set`和`map`集合映射标签就提供了一个`sort`属性，通过这个属性可以配置用于在Java中排序的字段，或者数据库本身也提供了一个用于排序的SQL `order-by`属性（[\[1\]](#)）。无论哪一种情况，当遍历集合的内容时，都能以特定的顺序获得其内容。

`LIST_POS`字段中的值总是与传递给`tracks.get()`方法的参数值相同，该方法用于获取`tracks`列表中某个特定位置上的值。

[\[1\]](#) 用`order-by`属性和SQL对集合进行排序的功能，只有Java SDK 1.4或更高版本才支持，因为这需要用到1.4版本以后才有的`LinkedHashSet`或`LinkedHashMap`类。

扩充集合中的关联

好了，如果我们想把专辑中的曲目按一定的顺序排列，现在已经有了所需要的解决方法。那么，如果我们想保存其他信息，应该怎么办？例如曲目是在哪一张唱片中找到的？当我们映射一个关联的集合时，我们已经知道Hibernate会创建一个连接表来存储对象之间的关系。此外，我们也知道如何在ALBUM_TRACKS表中增加一个索引字段，以保存该集合元素的顺序。理想情况下，我们也想要能够扩充数据表的内容，以填入我们自己选择的信息，以便于保存专辑曲目的其他细节信息。

事实上，我们也可以做好这件事情，而且过程也很简单。

应该怎么做

到目前为止，我们已经看到过两种把数据表放进数据库模式的方法。第一种方法是明确地将Java对象的属性映射到数据表的字段；第二种方法是定义一个集合（值或关联的），并指定用于管理这个集合的数据表和字段。结果就是用这两种方法创建的数据表的使用是一样的。可以直接用该数据表的某些字段来映射对象的属性，而同时再用其他字段来管理集合的映射。这样可以让我们在实现以特定的顺序来

保存专辑曲目的同时，还能够通过增加其他细节来扩展数据表，以支持包含多张唱片的曲目专辑。

注意：这种灵活性要花点时间来习惯，不过有其道理所在。尤其是如果你想将对象映射到现有的数据库模式时。

我们需要一个新的数据对象（**AlbumTrack**），用于保存有关曲目在专辑中的使用方式的信息。由于我们已经实践了几个示例，了解了如何映射完整而独立存在的实体，因此**AlbumTrack**对象实在没有必要单独存在于**Album**实体的范围以外。这刚好是个机会，可以看一看组件映射是怎么回事。回想一下，在**Hibernate**的术语中，实体就是在持久化机制中独立存在的对象：它可以被创建、查询以及删除，这些操作都独立于其他任何对象，因此有其自身的持久化身份（表现为它必须具有**id**属性）。相反地，组件虽然是可以存储到数据库，也能从数据库检索出来的对象，但它只是其他实体的从属部分。在这个示例中，我们要把一组**AlbumTrack**对象列表定义成**Album**实体的组件。例5-4就是定义这一关系的**Album**类的映射文件。

例5-4: Album.hbm.xml（Album类的映射定义）

```
<?xml version="1.0"?>
<! DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.oreilly.hh.data.Album"table="ALBUM">
<meta attribute="class-description">
```


Represents an album in the music database, an organized list of tracks.

```
@author Jim Elliott (with help from Hibernate)
</meta>
<id column="ALBUM_ID" name="id" type="int">
<meta attribute="scope-set">protected</meta>
<generator class="native"/>
</id>
<property name="title" type="string">
<meta attribute="use-in-tostring">true</meta>
<column index="ALBUM_TITLE" name="TITLE" not-null="true"/>
</property>
<property name="numDiscs" type="integer"/>
<set name="artists" table="ALBUM_ARTISTS">
<key column="ALBUM_ID"/>
<many-to-many
class="com.oreilly.hh.data.Artist" column="ARTIST_ID"/>
</set>
<set name="comments" table="ALBUM_COMMENTS">
<key column="ALBUM_ID"/>
<element column="COMMENT" type="string"/>
</set>
<list name="tracks" table="ALBUM_TRACKS">❶
<meta attribute="use-in-tostring">true</meta>
<key column="ALBUM_ID"/>
<index column="LIST_POS"/>
<composite-element class="com.oreilly.hh.data.AlbumTrack">❷
<many-to-one class="com.oreilly.hh.data.Track" name="track">❸
<meta attribute="use-in-tostring">true</meta>
<column name="TRACK_ID"/>
</many-to-one>
<property name="disc" type="integer"/>❹
<property name="positionOnDisc" type="integer"/>❺
</composite-element>
</list>
<property name="added" type="date">
<meta attribute="field-description">
When the album was created</meta>
</property>
</class>
</hibernate-mapping>
```

在创建好Album.hbm.xml以后，还需要将它添加到hibernate.cfg.xml的映射资源列表中。打开src目录下的hibernate.cfg.xml文件，将例5-5中

用粗体突出显示的那一行添加到这个文件中。

例5-5: 将Album.hbm.xml添加到Hibernate配置文件中

```
<?xml version='1.0'encoding='utf-8'?>
<! DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
<session-factory>
.....
<mapping resource="com/oreilly/hh/data/Track.hbm.xml"/>
<mapping resource="com/oreilly/hh/data/Artist.hbm.xml"/>
<mapping resource="com/oreilly/hh/data/Album.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

多数内容都类似于以前看到过的映射配置，但是**tracks**列表的配置值得我们仔细探讨。关于讨论的内容，先回想一下刚才我们究竟想做什么。

我们想要让专辑里的曲目列表保持一定的顺序，同时又能为每个曲目增加一些额外的信息，以指出曲目所属的唱片（专辑包括多张唱片的情况），以及该曲目在唱片中的位置。这种关系的概念如图5-1的中部内容所示。专辑和曲目之间的关联是由**"AlbumTrack"**对象作为媒介而体现的，这个对象新增加了唱片和位置信息，并保证曲目以正确的顺序排列。曲目对象本身的模型我们已经很熟悉了（此图中，为了保持简单，我们删去了艺人和评论信息），这个模型正是我们在专辑映射文档中需要使用的（例5-4）。让我们讨论其中的细节吧。稍后，

我们会介绍Hibernate如何将这样的映射配置规定转换成Java代码（图5-1的底部部分）和数据库模式（图5-1的顶部部分）。

好吧，有了这一概念性框架的提示和描述，我们接下来就看看例5-4的细节内容：

❶如果拿前一章的集合映射定义和这里的列表定义进行比较，你会发现它们之间存在很多相似之处。它看起来甚至更像例5-2，只是关联映射定义已经移到一个新的composite-element映射元素内部了。

❷这个元素引入新的AlbumTrack对象，我们用它来分组唱片、位置以及组织专辑曲目所需要的Track链接。

❸此外，AlbumTrack和Track之间的关联是多对一的（不是多对多的映射，因为专辑通常包含数个曲目，而特定曲目文件可能在几个专辑之间共享）：如果我们为了节省磁盘空间，几个AlbumTrack对象（来自不同专辑）可能会引用相同的Track文件，但每一个AlbumTrack对象只与一个Track相关。包含AlbumTrack的list标签隐含是一对多的关系（如果这些数据建模概念让你很困惑，现在不用太花费精力去弄懂这些，源代码和数据库模式马上就会出来，希望有助于你明白这到底在干什么）。

好了，继续从整体上考虑一下这个新的composite-element元素定义。这个元素指出我们想要用一个新的AlbumTrack类作为Album数据

bean的曲目列表中的值。composite-element标签的主体定义了AlbumTrack的各个属性，把专辑中一个曲目的所有信息都汇集在这儿了。这些嵌套属性的映射语法和外层Album自身属性的映射语法并没有什么不同，甚至还能包含自己的嵌套复合元素、集合或meta元素（如此处所示）。这让我们在建立细粒度的映射上具有相当大的灵活性，同时也保留了一定良好程度的面向对象的封装。

在我们的复合AlbumTrack映射中，我们要记录 and 实际Track之间的关联（刚才介绍的many-to-one元素），以及该曲目在专辑中的播放位置。

④复合映射也需要保存曲目所属的唱片的编号。

⑤最后也要保存该曲目在唱片中的位置（例如第2号唱片的第3首曲目）。

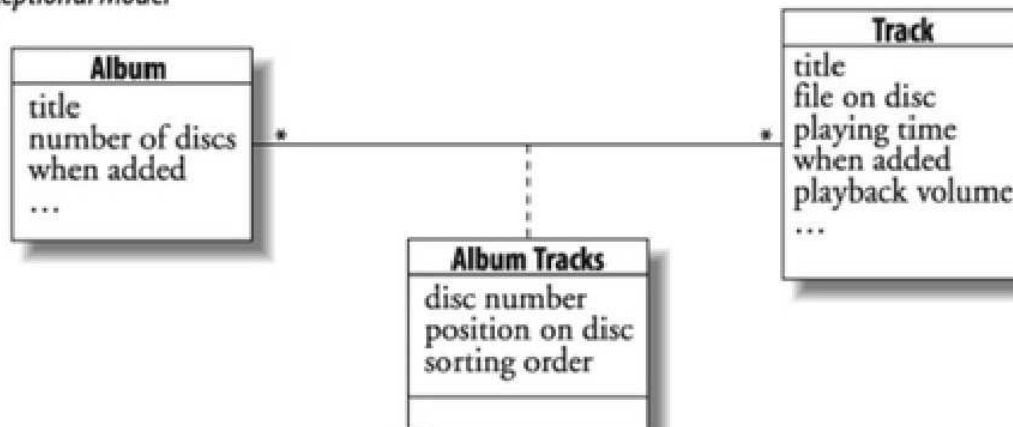
这一映射取得了我们先前的既定目标，也就是可以将任意的信息附加到关联集合中。

组件类本身的源代码如例5-6所示，它有助于阐明相关讨论。可以对比一下该源代码和它的图形表示图5-1底部。

Database Schema



Conceptional Model



Java Objects (Data Beans)

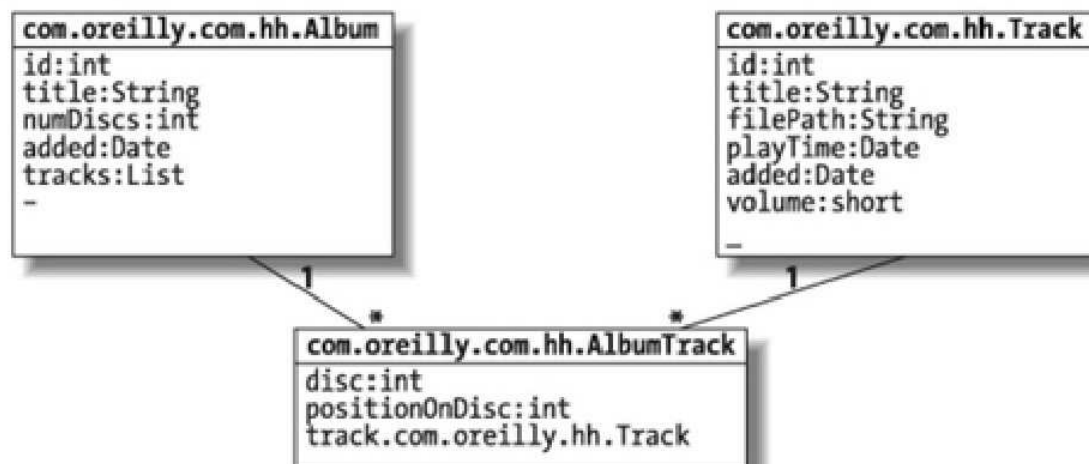


图 5-1 表示专辑曲目所涉及的数据表、概念、以及对象的模型

可以看到，我们选择TRACK_ID的字段名称作为链接到TRACK表的多对一（many-to-one）关联的连接字段。这样的处理在前面已经遇到过很多次了，但之前都不需要独立成一行。这里值得讨论一下这样选择的原因。没有这条指令，Hibernate将只会用属性名（track）作为字段名称。你的字段可以使用任何名称，但《Java Database Best Practices》一书鼓励我们把外键（foreign key）字段命名成和其引用的原来数据表里的主键（primary key）的名称相同。这有助于数据建模工具识别并显示外键所代表的“自然连接”（natural join），也可以让人更容易理解和使用数据。出于这种考虑，我们就将数据表名称作为主键字段名称的一部分。

发生了什么事

我原本以为，由于选择使用composite元素来封装扩充过的曲目列表，就得自己编写AlbumTrack类的Java源代码，而这会超出代码生成工具能力所及的范围。但是出乎意料的是，当我试着执行ant codegen命令，想看看有什么错误信息会出现时，没想到这个命令居然报告成功，源代码目录下出现了Album.java和AlbumTrack.java这两个文件！

注意：偶尔证明是错的也不为过。

这时，我再回过头来为组件内曲目的多对一映射新增了一个use-in-tostring的meta元素。我不确信这是否行得通，因为我在参考文档中惟一找到的使用示例都是附加在property标签以内。但是居然行得通，这和我原来希望的一样。

Hibernate最佳实践鼓励我们使用细粒度的（fine-grained）持久化类，再将它们映射为组件。代码生成工具可以轻易地根据映射文档来创建持久化类的源代码，绝对没有借口而对这一建议视而不见。例5-6演示了为嵌套的组件映射生成的源代码。

例5-6：为AlbumTrack.java生成的代码

```
package com.oreilly.hh.data;
//Generated Jun 21, 2007 11: 11: 48 AM by Hibernate Tools 3.2.0.b9
/**
 *Represents an album in the music database, an organized list of
 tracks.
 *@author Jim Elliott (with help from Hibernate)
 */
public class AlbumTrack implements java.io.Serializable{
    private Track track;
    private Integer disc;
    private Integer positionOnDisc;
    public AlbumTrack () {
    }
    public AlbumTrack (Track track, Integer disc, Integer
positionOnDisc) {
        this.track=track;
        this.disc=disc;
        this.positionOnDisc=positionOnDisc;
    }
    public Track getTrack () {
        return this.track;
    }
    public void setTrack (Track track) {
        this.track=track;
    }
}
```

```

    }
    public Integer getDisc () {
        return this.disc;
    }
    public void setDisc (Integer disc) {
        this.disc=disc;
    }
    public Integer getPositionOnDisc () {
        return this.positionOnDisc;
    }
    public void setPositionOnDisc (Integer positionOnDisc) {
        this.positionOnDisc=positionOnDisc;
    }
    /**
     *toString
     *@return String
     */
    public String toString () {
        StringBuffer buffer=new StringBuffer ();
        buffer.append (getClass ().getName ()) .append ("@") .append (
            Integer.toHexString (hashCode ()) ) .append ("[");
        buffer.append ("track") .append ("='") .append (getTrack
() ) .append ("");
        buffer.append ("]");
        return buffer.toString ();
    }
}

```

这段代码看起来和前几章为实体生成的代码差不多，但是此处少了一个id属性，这是有道理的。组件类不需要标识符字段，也不需要实现任何特殊接口。这个类和Album类共享同样的JavaDoc，而在Album类中就使用了该组件类。Album类的源代码是典型的代码生成工具生成的实体，所以这里不再赘述。

现在，我们可以通过ant schema命令为这些新的映射文档建立数据库模式了。例5-7演示了模式创建结果的重点内容，这就是图5-1顶端模式模型的具体HSQLDB表示。

例5-7: 由新的Album映射所增加的数据库模式

```
.....
[hibernatetool]create table ALBUM (ALBUM_ID integer generated by
default
as identity (start with 1) , TITLE varchar (255) not null,
numDiscs integer, added date, primary key (ALBUM_ID) ) ;
.....
[hibernatetool]create table ALBUM_ARTISTS (ALBUM_ID integer not
null,
ARTIST_ID integer not null,
primary key (ALBUM_ID, ARTIST_ID) ) ;
.....
[hibernatetool]create table ALBUM_COMMENTS (ALBUM_ID integer not
null,
COMMENT varchar (255) ) ;
.....
[hibernatetool]create table ALBUM_TRACKS (ALBUM_ID integer not
null,
TRACK_ID integer, disc integer, positionOnDisc integer, LIST_POS
integer not null,
primary key (ALBUM_ID, LIST_POS) ) ;
.....
[hibernatetool]create index ALBUM_TITLE on ALBUM (TITLE) ; .....
[hibernatetool]alter table ALBUM_ARTISTS add constraint
FK7BA403FC620962DF
foreign key (ARTIST_ID) references ARTIST;
[hibernatetool]alter table ALBUM_ARTISTS add constraint
FK7BA403FC3C553835
foreign key (ALBUM_ID) references ALBUM;
[hibernatetool]alter table ALBUM_COMMENTS add constraint
FK1E2C21E43C553835
foreign key (ALBUM_ID) references ALBUM;
[hibernatetool]alter table ALBUM_TRACKS add constraint
FKD1CBBC782DCBFAB5
foreign key (TRACK_ID) references TRACK;
[hibernatetool]alter table ALBUM_TRACKS add constraint
FKD1CBBC783C553835
foreign key (ALBUM_ID) references ALBUM;
.....
```

你可能发现, 对数据库模式做一些关键的修改会导致Hibernate或HSQLDB驱动程序发生问题。当我换用这种新方法来建立专辑曲目的

映射时，遇到了麻烦，因为第一组映射会建立一些数据库约束，而 **Hibernate** 在试图重新建立模式时并不知道应该先删除这些约束。这样就不能继续删除和重新创建某些数据表了。如果你也遇到了这种问题，可以先删除数据库文件（**data** 目录下的 **music.script** 文件），再从头开始，应该就没有问题了。针对这些情况，**Hibernate** 最新版本的健壮性似乎有所增强。

图5-2展示了 **HSQldb** 图形管理界面中扩展后的数据库模式。

你可能会问，究竟为什么要用这么一个单独的 **Track** 类，而不是直接把所有信息都放在扩充后的 **AlbumTrack** 集合中。答案很简单，并不是所有曲目都属于某个专辑，有些也许是单曲、下载而来的或其他单独的曲目。既然我们已经用一个单独的数据表来记录这些数据，所以再在 **AlbumTracks** 表中重复其内容而非建立与 **Track** 表的关联，将是一种很糟糕的设计。采用这种方法（我自己的音乐数据库就是用这种方法），还有另一个微妙的优点：这样的结构能让我们在多个专辑中共享同一个曲目。如果有专辑、精选集以及某个或多个年代的选集都出现了相同曲目，把这些曲目集都链接到相同的曲目就可以节省磁盘空间。

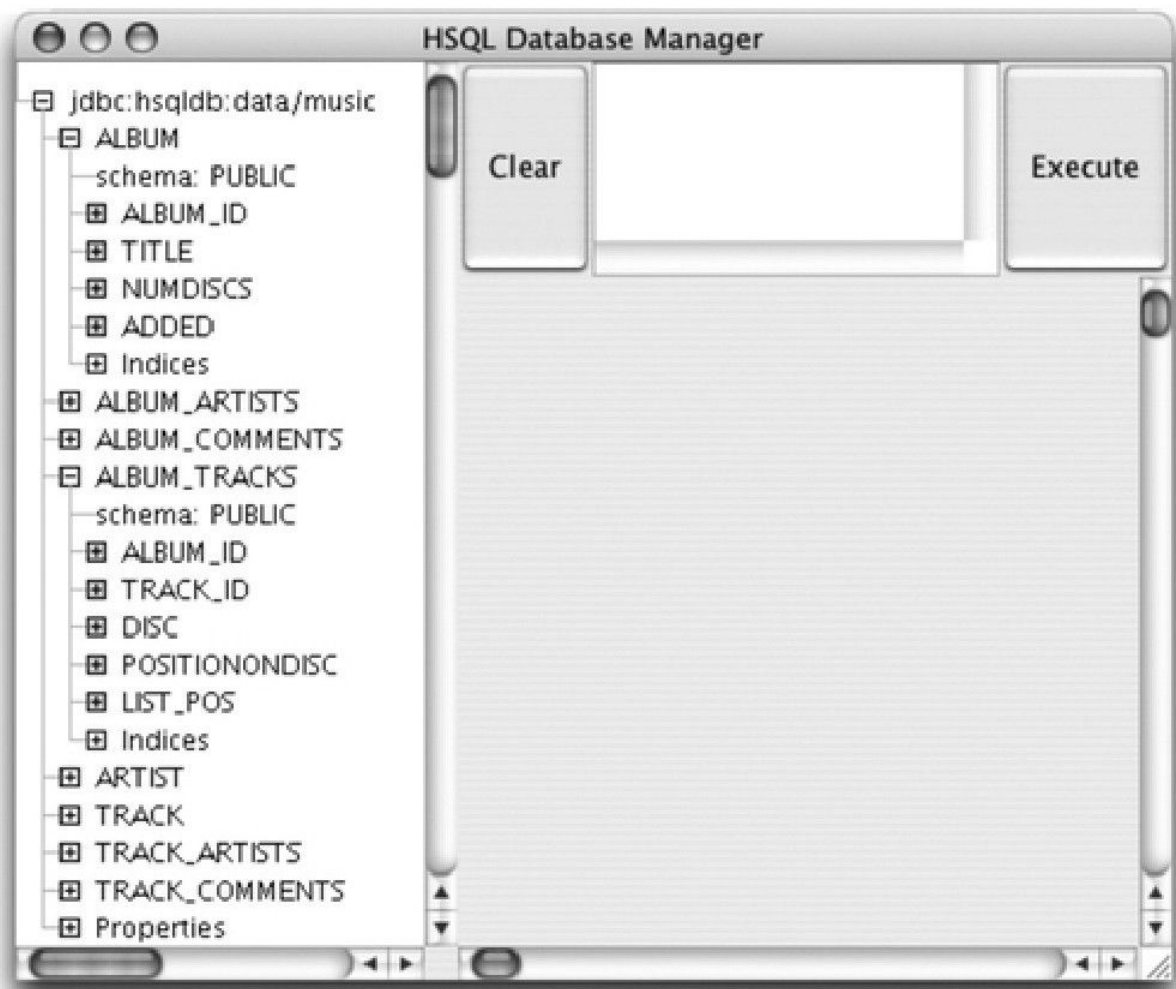


图 5-2 专辑相关数据表的模式

ALBUM_TRACK模式中另一个值得注意的地方是它没有明显的ID字段。如果查看例5-7中Hibernate为ALBUM_TRACK生成的模式定义，可以看到主键是通过primary key (ALBUM_ID, LIST_POS)这样的语句定义的。Hibernate已经知道，根据我们在Album.hbm.xml中要求的映射关系，ALBUM_TRACK表中的某一行可以由Album（专辑）的ID和曲目在列表中的索引而惟一确定，所以Hibernate为这个表建立了一个

复合主键（composite key）。对于这一优化，我们不需要过多讨论。还要注意，这些列中有一个类型是AlbumTrack类的属性，而其他列则不是。在第7章中，我们将以另一种稍微不同的方式来建立这一关系模型。

我们来看看示例程序代码，以了解如何使用这些新的数据对象。例5-8演示的类会创建一个专辑记录和一系列曲目，然后通过配置好的toString（）方法来打印输出测试调试数据。

例5-8: AlbumTest.java的源代码

```
package com.oreilly.hh;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import com.oreilly.hh.data.*;
import java.sql.Time;
import java.util.*;
/**
 *Create sample album data, letting Hibernate persist it for us.*/
public class AlbumTest{
/**
 *Quick and dirty helper method to handle repetitive portion of
creating
 *album tracks.A real implementation would have much more
flexibility.
 */
private static void addAlbumTrack (Album album, String title,
String file,
Time length, Artist artist, int disc,
int positionOnDisc, Session session) {❶
Track track=new Track (title, file, length, new HashSet<Artist>
() ,
new Date () , (short) 0, new HashSet<String> () ) ;
track.getArtists () .add (artist) ; ❷
session.save (track) ;
album.getTracks () .add (new AlbumTrack (track, disc,
positionOnDisc) ) ; ❸}
```

```

public static void main (String args[]) throws Exception{
//Create a configuration based on the properties file we've put
//in the standard place.
Configuration config=new Configuration () ;
config.configure () ;
//Get the session factory we can use for persistence
SessionFactory sessionFactory=config.buildSessionFactory () ;
//Ask for a session using the JDBC information we've configured
Session session=sessionFactory.openSession () ;
Transaction tx=null;
try{
//Create some data and persist it
tx=session.beginTransaction () ;
Artist artist=CreateTest.getArtist ("Martin L.Gore", true,
session) ;
Album album=new Album ("Counterfeit e.p.", 1,
new HashSet<Artist> () , new HashSet<String> () ,
new ArrayList<AlbumTrack> (5) , new Date () ) ;
album.getArtists () .add (artist) ;
session.save (album) ;
addAlbumTrack (album, "Compulsion", "vol1/album83/track01.mp3",
Time.valueOf ("00: 05: 29") , artist, 1, 1, session) ;
addAlbumTrack (album, "In a Manner of Speaking",
"vol1/album83/track02.mp3", Time.valueOf ("00: 04: 21") ,
artist, 1, 2, session) ;
addAlbumTrack (album, "Smile in the Crowd",
"vol1/album83/track03.mp3", Time.valueOf ("00: 05: 06") ,
artist, 1, 3, session) ;
addAlbumTrack (album, "Gone", "vol1/album83/track04.mp3",
Time.valueOf ("00: 03: 32") , artist, 1, 4, session) ;
addAlbumTrack (album, "Never Turn Your Back on Mother Earth",
"vol1/album83/track05.mp3", Time.valueOf ("00: 03: 07") ,
artist, 1, 5, session) ;
addAlbumTrack (album, "Motherless
Child", "vol1/album83/track06.mp3",
Time.valueOf ("00: 03: 32") , artist, 1, 6, session) ;
System.out.println (album) ;
//We're done; make our changes permanent
tx.commit () ;
//This commented out section is for experimenting with deletions.
//tx=session.beginTransaction () ;
//album.getTracks () .remove (1) ;
//session.update (album) ;
//tx.commit () ;
//tx=session.beginTransaction () ;
//session.delete (album) ;
//tx.commit () ;
}catch (Exception e) {

```

```
if (tx != null) {  
    //Something went wrong; discard all partial changes  
    tx.rollback ();  
}  
throw new Exception ("Transaction failed", e) ;  
}finally{  
    //No matter what, close the session  
    session.close () ;  
}  
//Clean up after ourselves  
sessionFactory.close () ;  
}  
}
```

❶首先，`addAlbumTrack ()` 方法会根据指定的参数创建一个`Track`对象，并将之持久保存。

❷其次，将新的曲目和一个`Artist`对象建立关联。

❸最后，将曲目对象加到`Album`内，记录所属的唱片以及在该唱片中的位置。

在这个示例中，我们创建了一个只有一张唱片的专辑。虽然这种快速而简单的方法并不能处理其他各种应用需求，但这样可以让示例更简洁一些。

为了调用这个类，还需要在`build.xml`的末尾增加一个新的`ant`构建目标，将例5-9的内容添加到该文件的结尾处（当然，应该在`project`元素的内部）。

例5-9: 运行`AlbumTest`类需要的`ant`构建目标

```
<target name="atest" description="Creates and persists some album
data"
  depends="compile">
  <java classname="com.oreilly.hh.AlbumTest" fork="true">
  <classpath refid="project.class.path"/>
  </java>
  </target>
```

准备好以后，假定已经生成了数据库模式，接着就依次执行ant ctest和ant atest（先运行ctest并不是必需的，但是，测试开始时有些额外的数据，会让专辑数据变得更有趣些。回想一下，你也可以只用一行命令来运行这些构建目标：ant ctest atest。如果想在开始时先把数据库的内容清除掉，则可以执行ant schema ctest atest）。这个命令产生的调试输出信息如例5-10所示。虽然有些难懂，但是应该可以看出来这段代码创建好了专辑和曲目数据，而且还保留了曲目的顺序。

例5-10：运行专辑测试的输出

```
atest:
[java]com.oreilly.hh.data.Album@5bcf3a[title='Counterfeit
e.p.'tracks='[
  com.oreilly.hh.data.AlbumTrack@6a346a[track='com.oreilly.hh.data.
Track@973271[
  title='Compulsion'volume='Volume[left=100,
right=100]'sourceMedia='CD']], c
om.oreilly.hh.data.AlbumTrack@8e0e1[track='com.oreilly.hh.data.Tr
ack@e3f8b9[ti
tle='In a Manner of Speaking'volume='Volume[left=100,
right=100]'sourceMedia='
CD']],
com.oreilly.hh.data.AlbumTrack@de59f0[track='com.oreilly.hh.data.Tra
c
k@e2d159[title='Smile in the Crowd'volume='Volume[left=100,
right=100]'source
Media='CD']],
com.oreilly.hh.data.AlbumTrack@1e5a36[track='com.oreilly.hh.da
```

```
ta.Track@b4bb65[title='Gone'volume='Volume[left=100,
right=100]'sourceMedia='
CD']'],
com.oreilly.hh.data.AlbumTrack@7b1683[track='com.oreilly.hh.data.Tra
c
k@3171e[title='Never Turn Your Back on Mother
Earth'volume='Volume[left=100, r
ight=100]'sourceMedia='CD']'],
com.oreilly.hh.data.AlbumTrack@e2e4d7[track='
com.oreilly.hh.data.Track@1dfc6e[title='Motherless
Child'volume='Volume[left=1
00, right=100]'sourceMedia='CD']']']']
```

如果执行原来的查询测试，会同时看到新旧数据，如例5-11所示。

例5-11： 无论是否来自专辑，所有曲目的播放时间都小于7分钟

```
%ant qtest
Buildfile: build.xml
.....
qtest:
[java]Track: "Russian Trance" (PPK) 00: 03: 30
[java]Track: "Video Killed the Radio Star" (The Buggles) 00: 03:
49[java]Track: "Gravity's Angel" (Laurie Anderson) 00: 06: 06
[java]Track: "Adagio for Strings (Ferry Corsten Remix) " (Ferry
Corsten,
Samuel Barber, William Orbit) 00: 06: 35
[java]Track: "Test Tone 1"00: 00: 10
[java]Comment: Pink noise to test equalization
[java]Track: "Compulsion" (Martin L.Gore) 00: 05: 29
[java]Track: "In a Manner of Speaking" (Martin L.Gore) 00: 04:
21[java]Track: "Smile in the Crowd" (Martin L.Gore) 00: 05: 06
[java]Track: "Gone" (Martin L.Gore) 00: 03: 32
[java]Track: "Never Turn Your Back on Mother Earth" (Martin
L.Gore)
00: 03: 07
[java]Track: "Motherless Child" (Martin L.Gore) 00: 03: 32
BUILD SUCCESSFUL
Total time: 2 seconds
```

最后，图5-3显示了在HSQLDB界面中检索ALBUM_TRACKS表内容的查询。

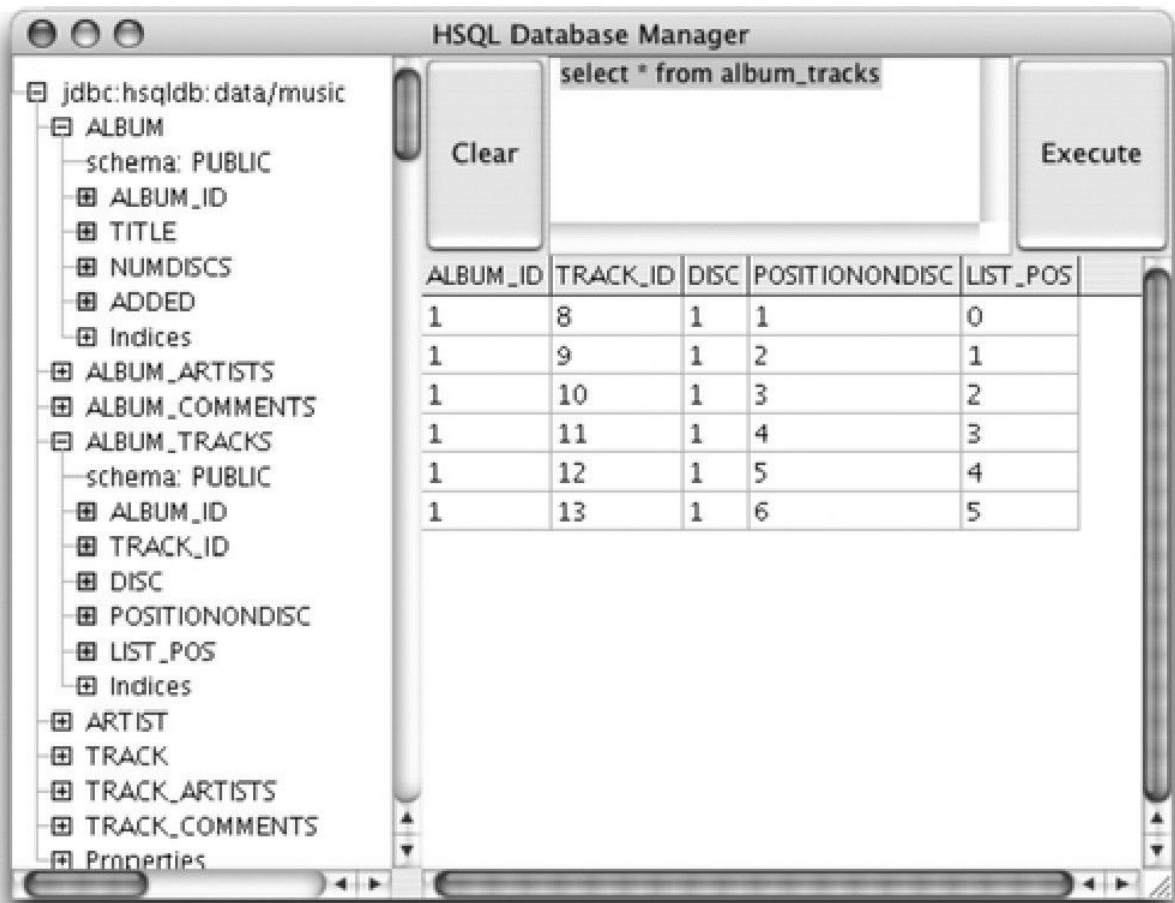


图 5-3 内容经过扩充后的关联集合

其他

删除集合元素、重新排列以及其他对这些相互关联的信息的处理？这些处理都是自动支持的，下一节将会演示这些内容。

关联的生命周期

Hibernate完全负责ALBUM_TRACKS表的管理，当为Album bean的tracks属性增加或删除数据项时，Hibernate会自动向ALBUM_TRACKS表中增加或删除相应的记录行（必要时会重新计算LIST_POS值）。可以写个测试程序进行测试，从我们的测试专辑中删除第2个曲目，然后看看结果如何。一种快速而简单的做法就是将下列4行代码（如例5-12所示）添加到例5-8中已有的tx.commit（）那一行的后面，然后执行ant schema ctest attest db命令。

例5-12：删除专辑的第2个曲目

```
tx=session.beginTransaction();
album.getTracks().remove(1);
session.update(album);
tx.commit();
```

这样做会改变ALBUM_TRACKS表中的内容，如图5-4所示（和图5-3的原始内容相比较）。第2条记录已经被删除（记住，Java列表元素的索引值是从0开始的），而LIST_POS也做了调整以保证其连续性，以便与列表元素的索引可以相对应（调用tracks.get（）所用的参数值）。

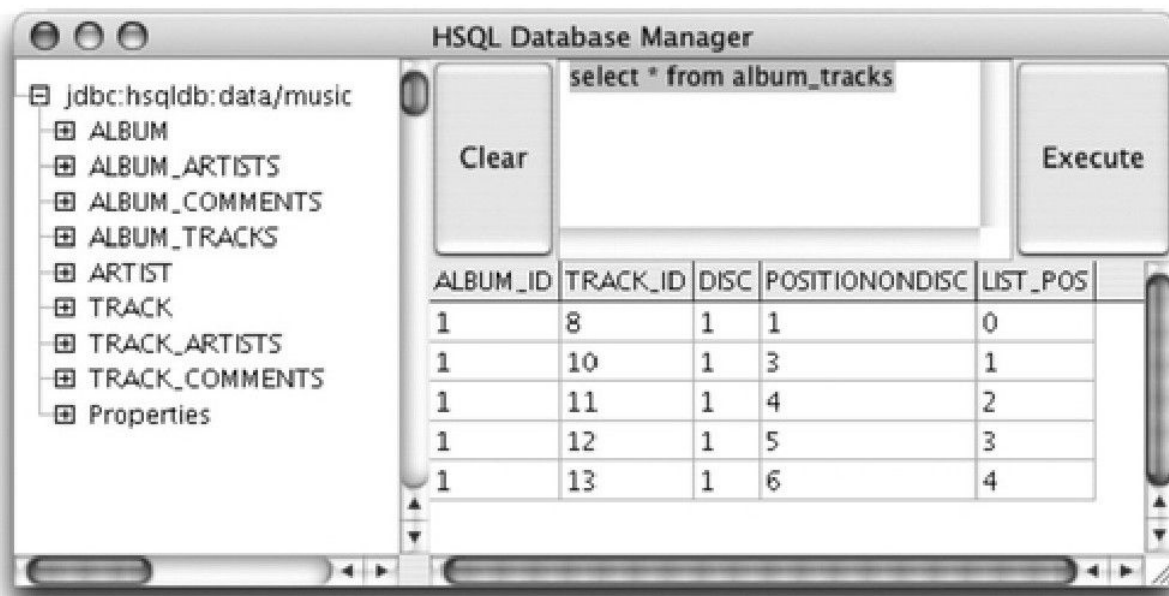


图 5-4 删除专辑的第2个曲目后的专辑曲目关联

之所以这样，是因为Hibernate明白这个列表是由Album表的记录“拥有”的，而这两个对象的“生命周期”（lifecycle）彼此紧密相连。设想一下，如果整个Album表的内容都被删除会发生什么事：ALBUM_TRACKS里所有相关联的记录也会跟着被删除（如果你不信，可以修改测试程序来试试看）。这么一想，生命周期的概念就变得更为清晰了。

ALBUM表和TRACK表之间的关系就不同了。曲目有时会 and 专辑关联，但有时也是独立的。从列表中删除一个曲目，就会导致清除ALBUM_TRACKS表中相应的一行记录，专辑和曲目之间的关联也随之消失，但并不会删除TRACK表中的记录行，所以这么做也不会删除持久化Track对象本身。同样地，删除Album对象会把集合中所有的关

联都清除，但所有实际的Track对象都不受影响。我们的代码只是负责在适当的时机进行这些操作（或许在向用户咨询以后，说不定有些曲目记录可以在多个专辑中共享，如前所述）。

如果我们不需要在专辑之间共享相同曲目的灵活性（对于压缩的音频文件的体积来说，其占用的磁盘空间最近相当便宜），也可以让Hibernate以管理ALBUM_TRACKS集合的相同方式来管理专辑的TRACK记录。Hibernate当然不会以为它应该这么做，因为Track和Album对象能彼此独立存在，但是我们可以在专辑映射文档中为二者之间建立生命周期关系。

注意：现在，已经知道有可以自动化处理这些的方式，可能不会再感到吃惊了吧。

应该怎么做

例5-13展示了我们对Album.hbm.xml里的tracks属性映射做出的修改（以粗体表示）。

例5-13：为专辑和它的曲目建立生命周期关系

```
<list name="tracks" table="ALBUM_TRACKS" cascade="all">
  <meta attribute="use-in-tostring">true</meta>
  <key column="ALBUM_ID"/>
  <index column="LIST_POS"/>
  <composite-element class="com.oreilly.hh.AlbumTrack">
    <many-to-one class="com.oreilly.hh.Track" name="track"
```

```
cascade="all">
<meta attribute="use-in-tostring">true</meta>
<column name="TRACK_ID"/>
</many-to-one>
<property name="disc" type="integer"/>
<property name="positionOnDisc" type="integer"/>
</composite-element>
</list>
```

`cascade`属性用于告诉Hibernate，需要将“父”（parent）对象上施加的操作也应用到它的“子”（child）或“依赖”（dependent）对象上。这一属性适用于所有形式的集合和关联。它有几种可供选择的预设值，最常见的是`none`（默认值）、`save-update`、`delete`以及`all`（组合了`save-update`和`delete`）。还可以通过在`hibernate-mapping`标签内部提供一个`default-cascade`属性，将整个映射文档范围内的默认值从`none`变为`save-update`。

就此例而言，我们希望专辑所包含的曲目都由专辑自动管理，这样，当删除某个专辑时，它的曲目也会随之删除。注意，我们必须为`tracks`集合和组成它的`track`元素都应用`cascade`属性，才能做到这一点。此外，将`cascade`属性设置为`all`时，就不用再明确保存我们为专辑而创建的任何`Track`对象，也就是例5-8中的`addAlbumTrack()`方法不再需要以下这行：

```
session.save(track);
```

通过告诉Hibernate让它负责维护专辑及其曲目之间的关联关系，就可以让Hibernate在曲目被加进专辑时自动将曲目对象（Track对象）持久保存，同时在删除专辑时也可以删除相关的所有曲目。

Hibernate对生命周期关系的管理并不是十分安全的（fool-proof），或许更准确地说，它并非是无所不包的。例如，如果使用Collections接口的方法从Album的tracks属性中删除一个Track对象，这样会打断Album和Track之间的关联，但实际上并不会删除那个Track对象对应的记录。即使以后把整个Album删除掉，那个Track还会保留着，因为删除Album的时候，原来被删除的那个Track已经没有和被删除的Album之间有关联了。适当修改AlbumTest.java就能做一些这类实验，看看数据表中最后的结果！

事实上，在某些特定的情况下，Hibernate可以负责处理这种级别的细节。只要在父子关系中使用多对一（many-to-one）映射，就可以将映射的cascade属性标识为delete-orphan。相关的更多细节可以查阅Hibernate在线参考手册的“传播性持久化”（Transitive persistence（[\[1\]](#)））一节。

将这种信息登记处理委托给映射层实在方便，这样你就能把精力集中在更为抽象和重要的任务上，所以在时机适当时值得使用。这会让人想起Java那令人信服的垃圾收集机制所带来的程序员的解放，但是也有一定的局限性，例如，无法通过可达性分析（reachability

analysis) 而明确地知道什么时候已经完成了数据的持久化。你得调用 `delete ()`，并建立生命周期连接，才可以在代码中指明这一点。灵活性和自动化的简单性之间的得失是由你决定的，取决于数据的本质和项目的需要。

[1]

http://www.hibernate.org/hib_docs/v3/reference/en/html/objectstate.html#objectstate-transitive.

自身关联

对象和数据表也可以关联到它们自身，这种关联叫做自身关联（**reflexive association**）。这种关联用于支持持久化递归结构的数据，例如树状结构，这种结构中的节点之间会彼此互相链接。数据库表中如果存储了这种关系的数据，要以SQL查询接口来检索这些数据将非常困难。所幸，将这种关联映射到Java对象后，处理就变得更容易理解和自然了。

在我们的音乐数据库中可能使用自身链接（**reflexive link**）的方式之一是让艺人可以有替换姓名（**alternate name**）。这一功能的用处可能会超出你的预期，因为如果让用户根据他们的思维方式来决定查找"The Smiths"或"Smiths, The"，有了替换姓名就容易多了，只要一点点程序代码，而且实现方式与编程语言无关。

注意：我是指人类语言，英语、西班牙语或其他语言。在数据中放入这类链接，而不是试图去编写难以理解的程序代码来猜测何时应该调换艺人的姓名。

应该怎么做

需要做的事就是在Artist.hbm.xml里为Artist映射新增另一个字段，以建立一个指向Artist的链接。例5-14演示了一种配置方法。

例5-14：在Artist类中建立自身关联

```
<many-to-one  
name="actualArtist"class="com.oreilly.hh.data.Artist">  
  <meta attribute="use-in-tostring">true</meta>  
</many-to-one>
```

这个示例配置了一个actualArtist属性，在建立一个艺人的替换姓名时，我们可以将其设置为“实质的”Artist记录的id值。例如，“The Smiths”记录的id值可能是5，而它的actualArtist字段应该是null，因为这个记录是实质的艺人记录。然后，我们随时都可以用“Smiths, The”这样的姓名创建“别名”Artist记录，并将这个别名记录里的actualArtist字段设置为5，以指向实质的记录。

有时，作为外键的字段不能按照其链接到的主键字段的名称来命名，自身链接就是这样的一个例子。我们将ARTIST表中的一行数据关联到ARTIST表中的另一行，当然，数据表中已经有名为ARTIST_ID的字段了。

为什么要将这种关联设置为多对一（many-to-one）类型的？也许会有很多别名记录会指向某个特定的实质Artist记录。所以，每个别名

记录都必须保存实际艺人记录的id，才能视其为替换姓名。用数据建模语言来说，这就是一种多对一关系。

查找艺人的代码只需要在返回前检查actualArtist属性。如果是null，一切都没有问题；否则，就应该返回actualArtist属性指向的记录。例5-15扩展了CreateTest中的getArtist () 方法，以支持这种新功能（新增处以粗体字表示）。注意，Artist构造函数多了一个用于设置actualArtist的新参数，所以，即使我们不用显示替换姓名，也得更新CreateTest中其他调用这个构造函数的各个地方。

例5-15：支持替换姓名解析的艺人查询方法

```
public static Artist getArtist (String name, boolean create,
Session session) {
    Query query=session.getNamedQuery
("com.oreilly.hh.artistByName");
    query.setString ("name", name);
    Artist found= (Artist) query.uniqueResult ();
    if (found==null&&create) {
        found=new Artist (name, new HashSet () , null);
        session.save (found);
    }
    if (found!=null&&found.getActualArtist () !=null) {
        return found.getActualArtist ();
    }
    return found;
}
```

希望这一章能让你感受到Hibernate中关联和集合的丰富而强大的功能。很明显，你可以结合这些功能，层层嵌套，当中的变化之多，恐怕我们难以在像这样的一本书中解释清楚。

好消息是**Hibernate**似乎已经相当完善，足以应付实际应用中可能会遇到的各种关联关系，甚至为你完成建立数据类和数据库模式，这样辛苦繁重的工作。当我开始创建这些示例时，**Hibernate**工作的有效性和深远性远远超过了我的预期。

第6章 自定义值类型

用户自定义类型

由附录A可见，**Hibernate**支持很多Java类型（既有简单的值类型，也有对象类型）。通过建立映射规范，甚至可以将非常复杂的、嵌套的对象结构持久化保存到任何数据表和字段内。既然**Hibernate**提供的功能强大而又灵活，你可能会问为什么**Hibernate**内建的类型支持还会不够用？

促使你使用**Hibernate**自定义类型的一种情况是，你想使用不同于**Hibernate**通常情况下会选择的SQL字段类型来保存某种特定的Java类型。**Hibernate**参考文档引用了一个例子，它将Java **BigInteger**类型的值持久化保存到**VARCHAR**字段。在某些为了兼容于传统数据库（**legacy database**）模式的情况下，就可能需要这么做。

另一种常见的自定义类型使用场合涉及枚举类型值的持久化。在Java 5以前，没有为枚举类型提供内建的语言支持。所以，尽管Joshua Bloch在他的《**Effective Java Programming Language Guide**》（Addison-Wesley）一书中提出的优秀的设计模式是事实上的标准，但**Hibernate**还是不知应该如何支持这一概念。在**Hibernate 3**之前，**Hibernate**就提

供了**PersistentEnum**接口，但这一接口与Java 5中引入的原生（**native**）枚举类型（**enum**）的支持并不适合，所以**PersistentEnum**对于Java 5的**enum**类型也没有什么用途。而且不幸的是，**Hibernate**现在还没有提供相应的替换解决办法，所以我们在这里将介绍如何利用**Hibernate**对自定义类型的支持来实现枚举类型的持久化。

另一种需要调整类型系统的场合是，你必须将一个属性值拆分成多个组成分部，并保存到多个数据库字段。也许，公司内部强制要求使用的可重用库中的**Address**对象是把ZIP+4码保存成了一个单独的字符串，但是你要整合的数据库却需要一个5位数字的字段和一个4位数字、其值可为**null**的字段，来保存这两部分的数据。或者，也有可能是相反的情况，需要将一个数据库字段拆分成多个属性。

所幸，像这种情况，**Hibernate**可以让你控制持久化映射的细节，使你在不得已时也能找到一种权宜之计。

注意：让简单的事情更容易，复杂的事情有可能。这样的精神要持续下去。

即使在某些不是严格必需的情况下，你也可能会想建立自定义值类型。如果你有一个复合类型（**composite type**），它在整个应用程序中的许多地方都用得到（向量、复数、地址等），你当然可以把会用到它的地方都映射成组件。但是，将映射细节封装在一个可共享的、

可重用的Java类内，比让映射细节在每个映射文档内到处传播可能更有价值。如此一来，如果映射细节因任何原因需要修改时，只需要修改一个类，而不用去寻找和调整众多特定组件的映射配置。

就上述所有使用场合而言，所需要做的任务就是告诉Hibernate一种新方法，让它知道如何对内存中特定类型的值及其数据库持久化保存形式做相互转化。

应该怎么做

Hibernate能让你在需要的情况下自行提供映射值的逻辑，办法就是实现以下两个接口之一：`org.hibernate.usertype.UserType`或`org.hibernate.usertype.CompositeUserType`。

Hibernate会为特定类型的值创建一个转化器（`translator`），而并非为它另外创建一种知道如何自行持久化的新类型的值，了解这一点很重要。换言之，以ZIP码为例，不是由ZIP码属性本身去实现`UserType`接口，而是我们要另外新建一个实现了`UserType`接口的类，然后在映射文档中将这个类指定为用于映射ZIP码属性的Java类型。因此，我认为“自定义类型”这个术语有些令人混淆。

我们来看一个具体的示例。如前所述，自定义类型的一个常见目标就是持久化枚举类型。虽然Hibernate本身没有提供对枚举类型的支

持，但是利用**UserType**机制可以容易地达到这一目的。之后，我们还会介绍一个更复杂的映射例子，它涉及多个属性和字段之间的映射。

定义一个持久化的枚举类型

枚举类型（**enumerated type**）是程序设计当中常见而且有用的抽象，可以让你从一组固定的命名选项中选取其值。枚举类型最初在 **Pascal** 中的表达形式相当不错，但 **C** 语言只实现了最基本的枚举类型（基本上只能把符号名称指定给某些可交换的整数值），结果早期的 **Java** 版本只保留了 **C** 语言的 **enum** 关键字，却一直没有实现它。一种名为“类型安全的枚举模式”（**typesafe enum pattern**）的面向对象的优秀方法逐步形成，并通过 **Joshua Bloch** 写的《**Effective Java**》一书而大受欢迎。这种枚举模式方法需要编写一大堆模板式的代码，但能够让你做各种有趣而且功能强大的事情。而 **Java 5** 规范带来的众多令人振奋的创新之一就是让 **enum** 关键字再度复活，这是一种获得类型安全的枚举功能的简单方法，并且不需要编写繁琐的模板代码。此外，还提供了其他好处。

注意：**C** 语言风格的数字型的枚举类型还是在 **Java** 中时常出现。
Sun API 中旧的部分就有很多。

不论枚举类型是用什么方法实现的，偶尔会需要将这种值持久化保存到数据库。虽然现在 **Java** 中已经有了标准的枚举类型，但

Hibernate并没有提供内建的支持。所以我们来看看如何用Hibernate的UserType接口来实现枚举值的持久化。

假设我们想要能够指明曲目来自何处，例如录音带、Vinyl、VHS影带、CD、广播、Internet下载网站以及数字音频流。（我们需要区分从Internet网站下载，还是从像Sirius或XM这类卫星无线电服务下载，或者区分是从无线电台还是从电视台下载。这实在会让人发疯，不过，用来说明这个重要概念倒是很适合的。）

不考虑持久化，类型安全的枚举类看起来可能类似于例6-1（已经对JavaDoc进行了压缩，以节约印刷空间。但是，如果从网站下载的话，格式是完整的）。

例6-1: SourceMedia.java（第一个类型安全的枚举类）

```
package com.oreilly.hh;
/**
 *This is a typesafe enumeration that identifies the media on
which an
 *item in our music database was obtained.
 */
public enum SourceMedia{
    /**Music obtained from magnetic cassette tape.*/
    CASSETTE ("Audio Cassette Tape") ,
    /**Music obtained from a vinyl record.*/
    VINYL ("Vinyl Record") ,
    /**Music obtained from VHS tapes.*/
    VHS ("VHS Videocassette tape") ,
    /**Music obtained from a broadcast.*/
    BROADCAST ("Analog Broadcast") ,
    /**Music obtained from a digital compact disc.*/
    CD ("Compact Disc") ,
    /**Music obtained as an Internet download.*/
```

```

    DOWNLOAD ("Internet Download") ,
    /**Music obtained from a digital audio stream.*/
    STREAM ("Digital Audio Stream") ;
    /**
     *Stores the human-readable description of this instance, by
    which it is
     *identified in the user interface.
     */
    private final String description;
    /**
     *Enum constructors are always private since they can only be
    accessed
     *through the enumeration mechanism.
     *
     *@param description human readable description of the source for
    the
     *audio, by which it is presented in the user interface.
     */
    private SourceMedia (String description) {
        this.description=description;
    }
    /**
     *Return the description associated with this enumeration
    instance.
     *
     *@return the human-readable description by which this value is
     *identified in the user interface.
     */
    public String getDescription () {
        return description;
    }
}

```

当然，使用Hibernate的妙处就是我们不需要修改通常的Java类，就可以增加持久化支持。即使我们原来在设计这一enum类型时还没有考虑持久化，我们以后也是照原样使用这个类型。由于现在不必考虑使用过时的PersistentEnum接口，那么定义一个可持久化的枚举类型与定义其他任意枚举类型就没有什么区别了。

注意：使用**PersistentEnum**接口需要修改欲持久化的枚举类型的定义。与**PersistentEnum**接口不符合《Effective Java》提出的可持久化枚举模式或Java 5 **enum**关键字相比，这可能是废除**PersistentEnum**接口的更大原因。

那么如何告诉**Hibernate**持久化保存我们的枚举类型的值呢？接下来就介绍这些。

使用自定义的类型映射

如本章介绍部分所述，我们打算创建一个枚举类，而且可以用Hibernate对它的值进行持久化。我们将这个新类命名为SourceMediaType。接下来再决定它需要实现UserType接口，还是CompositeUserType接口。Hibernate参考文档对此问题没有提供什么指导，但API文档对这两个接口名称所隐含的意义做了确认：只有在自定义类的实现想以命名属性的形式来暴露其内部结构，使其能在查询中被单独访问时（例如ZIP码的例子），才需要使用CompositeUserType接口。对于SourceMedia，实现简单的UserType就够用了。例6-2演示了符合我们需要的映射管理器（[\[1\]](#) 1）（mapping manager）的源代码。

例6-2：自定义类型映射处理器（SourceMediaType.java）

```
package com.oreilly.hh;
import java.io.Serializable;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;
import org.hibernate.Hibernate;
import org.hibernate.HibernateException;
import org.hibernate.usertype.UserType;
/**
 *Manages persistence for the{@link SourceMedia}typesafe
 enumeration.
 */
public class SourceMediaType implements UserType{
```

```

/**
 *Indicates whether objects managed by this type are mutable.
 *
 *@return<code>>false</code>, since enumeration instances are
immutable
 *singletons.
 */
public boolean isMutable () {
    return false;
}
/**
 *Return a deep copy of the persistent state, stopping at
 *entities and collections.
 *
 *@param value the object whose state is to be copied.
 *@return the same object, since enumeration instances are
singletons.
 */
public Object deepCopy (Object value) {
    return value;
}
/**
 *Compare two instances of the class mapped by this type for
persistence
 *"equality".
 *
 *@param x first object to be compared.
 *@param y second object to be compared.
 *@return<code>>true</code> iff both represent the same
SourceMedia type.
 *@throws ClassCastException if x or y isn't a{@link
SourceMedia}.
 */
public boolean equals (Object x, Object y) {
    //We can compare instances, since SourceMedia are immutable
singletons
    return (x==y) ;
}
/**
 *Determine the class that is returned by{@link#nullSafeGet}.
 *
 *@return{@link SourceMedia}, the actual type returned
 *by{@link#nullSafeGet}.
 */
public Class returnedClass () {
    return SourceMedia.class;
}
/**

```

*Determine the SQL type (s) of the column (s) used by this type mapping.

*

*@return a single VARCHAR column.

*/

public int[]sqlTypes () {❶

//Allocate a new array each time to protect against callers changing
//its contents.

int[]typeList=new int[1];
typeList[0]=Types.VARCHAR;

return typeList;
}

/**

*Retrieve an instance of the mapped class from a JDBC{@link
ResultSet}.

*

*@param rs the results from which the instance should be
retrieved.

*@param names the columns from which the instance should be
retrieved.

*@param owner the entity containing the value being retrieved.

*@return the retrieved{@link SourceMedia}value, or<code>null
</code>.

*@throws SQLException if there is a problem accessing the
database.

*/

public Object nullSafeGet (ResultSet rs, String[]names, Object
owner)

throws SQLException❷

{

//Start by looking up the value name

String name= (String) Hibernate.STRING.nullSafeGet (rs,
names[0]);

if (name==null) {

return null;

}

//Then find the corresponding enumeration value

try{

return SourceMedia.valueOf (name) ; ❸

}

catch (IllegalArgumentException e) {

throw new HibernateException ("Bad SourceMedia value: "+name,
e) ; ❹

}

}

/**

```

    *Write an instance of the mapped class to a{@link
PreparedStatement},
    *handling null values.
    *
    *@param st a JDBC prepared statement.
    *@param value the SourceMedia value to write.
    *@param index the parameter index within the prepared statement
at which
    *
    this value is to be written.
    *@throws SQLException if there is a problem accessing the
database.
    */
    public void nullSafeSet (PreparedStatement st, Object value, int
index)
    throws SQLException⑤
    {
    String name=null;
    if (value!=null)
    name= ( (SourceMedia) value) .toString () ;
    Hibernate.STRING.nullSafeSet (st, name, index) ;
    }
    /**
    *Reconstruct an object from the cacheable representation.At the
very least this
    *method should perform a deep copy if the type is mutable.
    (optional operation)
    *
    *@param cached the object to be cached
    *@param owner the owner of the cached object
    *@return a reconstructed object from the cacheable representation
    */
    public Object assemble (Serializable cached, Object owner) {
    return cached;
    }
    /**
    *Transform the object into its cacheable representation.At the
very least this
    *method should perform a deep copy if the type is mutable.That
may not be enough
    *for some implementations, however; for example, associations
must be cached as
    *identifier values. (optional operation)
    *
    *@param value the object to be cached
    *@return a cacheable representation of the object
    */
    public Serializable disassemble (Object value) {

```

```

        return (Serializable) value;
    }
    /**
     *Get a hashCode for an instance, consistent with
    persistence"equality".
     *@param x the instance whose hashCode is desired.
     */
    public int hashCode (Object x) {
        return x.hashCode ();
    }
    /**
     *During merge, replace the existing (target) value in the entity
    we are merging to
     *with a new (original) value from the detached entity we are
    merging.For immutable
     *objects, or null values, it is safe to simply return the first
    parameter.For
     *mutable objects, it is safe to return a copy of the first
    parameter.For objects
     *with component values, it might make sense to recursively
    replace component values.
     *
     *@param original the value from the detached entity being merged
     *@param target the value in the managed entity
     *@return the value to be merged
     */
    public Object replace (Object original, Object target, Object
owner)
        throws HibernateException
    {
        return original;
    }
}

```

这么长的一大段代码看起来有些令人畏惧，不过，不要担心。这个类中的所有方法都是由`UserType`接口指定的。我们的实现相当简洁明了，刚好符合我们所做的简单映射的需要。前三个方法没什么可讨论的，看看`JavaDoc`和代码内嵌的注释就够了，以下是对代码中一些有意思的地方进行的注释：

❶ `sqlTypes ()` 方法向Hibernate报告该自定义类型需要保存其值的列的数量，以及这些列的SQL类型。这里我们的类型使用一个 `VARCHAR` 类型的列。

因为API规定必须将这种信息作为数组返回，安全的编码实践要求在每次调用时都创建和返回一个新的数组，以防止恶意代码或错误代码可能操纵该数组的内容（Java不支持不可变数组。如果 `UserType` 接口声明这个方法返回 `Collection` 或 `List` 就好了，因为这些类型可以是不可变的）。

❷ 在 `nullSafeGet ()` 中，我们将数据结果转换为相应的 `MediaSource` 枚举值。因为我们知道这个值在数据库中保存为字符串，所以能够将实际的查询过程委托给Hibernate的工具方法，让它从数据库结果中加载字符串。在大多数情况下，都能够做这样的处理。

❸ 接着只需要使用枚举类型自己的实例查询功能。

❹ `HibernateException` 是当执行映射操作发生时，Hibernate抛出的一个 `Runtime-Exception`。我们这里“借用”了现成的 `Hibernate` 异常，因为可以认为问题与映射相关。如果我们想搞得花哨些，可以通过定义自己的异常类型来提供更多的细节，不过最好让自定义的异常类扩展 `HibernateException`，尤其是在像 `Spring` 这样的抽象框架中使用时，这样做更有意义（我们将在第13章介绍 `Spring`）。

❹另一个方向的映射是由`nullSafeSet()`处理的。我们再一次依靠Java与内建的`enum`机制将`SourceMedia`实例转换为相应的名称，接着使用Hibernate工具将这个字符串保存到数据库中。

在所有对值进行处理的方法中，很重要的一点就是编写代码的方法，如果任何一个参数为`null`时（这种情况经常发生），方法执行不会崩溃。方法名称前的“`nullSafe`”前缀就是对此的一种提醒，不过，即使是`equals()`方法也必须谨慎使用。盲目地使用`x.equals(y)`，当`x`为`null`时，就会让程序崩溃。

其他方法是一些普通的接口方法的实现，因为在处理不可变的单例对象（`immutable singleton`）时，虽然是枚举值，也应该避免在持久化管理时潜在的复杂性。

好吧，我们已经创建了一个自定义的类型持久化处理器，它并没有想象中的那么难。接下来就可以真正用它来持久化我们的枚举数据了。

应该怎么做

这实在是简单到令人尴尬。在有了值类、`SourceMedia`、持久化处理器和`SourceMediaType`以后，需要做的就是修改以前的映射文档，在

需要映射的地方，使用我们自定义的持久化管理器类而不是原先的原始值类型（raw value type）。

我们将通过一个例子，用媒体来源枚举类型来演示它的持久化。但是在深入这个例子以前，我们先稍候片刻，考虑一下如何让实现变得更加通用，以便在更大的项目中使用。

其他

如果需要持久化保存多个枚举类型呢？如果你曾经考虑过这个问题，你可能会认为这与例6-2在本质上没有什么不同，只不过例6-2只专注于持久化我们的SourceMedia枚举类型。代码中涉及类型的位置只有几处（如果忽略JavaDoc，甚至要更少）。为什么不能将枚举类型参数化，在一个单独的实现中就可以支持任何枚举类型，这样做不是更简单吗？

确实，这样做会相当简单，如果Hibernate要是内建了一种这样简单而灵活的机制，那就更好了。在Hibernate维基（[wiki](#)）上还有几种可供选择的方法（在几个不同的页面上，可能需要按主题类型浏览），或许我们应该采用Gavin King提出的Enhanced UserType（改进版的UserType）（在Java 5 EnumUserType（[\[2\]](#)）主题页），来作为我们非官方的“官方选择”（afficial choice），因为他是Hibernate诸多作者

之一。这种方法的功能看起来已经相当完整了，只是还没有轻率地付诸实用。不过，现在至少可以考虑一下将它与例6-2进行比较，看看在我们的解决方案更为通用化方面还能做哪些工作。

注意：我可以听到很多感慨“到时间了”！

但是现在，继续我们具体的示例，看看怎么样真正使用我们的枚举类型！

[1] 这里，原文中使用了映射管理器（**mapping manager**）以及自定义类型映射处理器（**custom type mapping handler**）这两个不同的词，容易让读者产生混淆。事实上，它们都是指同样的概念。

[2] <http://www.hibernate.org/272.html>.

使用持久化的枚举对象

你可能已经注意到，本章一开始并没有为**SourceMedia**类定义持久化映射。这是因为枚举类型是一个值，只能将它作为一个或多个实体的一部分而进行持久化保存，而不是自成一个实体。

因此，我们还没有做任何映射也就不奇怪了。当我们要实际使用持久化的枚举类型时，才需要这么做，这就是本节要介绍的内容。

应该怎么做

回想一下，我们想在自动唱片机系统中保存音乐曲目的来源媒介。也就是说，我们想在**Track**映射配置中使用**SourceMedia**这一枚举类型。可以简单地在**Track.hbm.xml**中的class定义内添加一个新的property标签，如例6-3所示。

例6-3：在Track映射文档中新增一个SourceMedia属性

```
.....
    <property name="volume" type="short">
      <meta attribute="field-description">How loud to play the track
    </meta>
    </property>
    <property
name="sourceMedia" type="com.oreilly.hh.SourceMediaType">
      <meta attribute="field-description">Media on which track was
obtained</meta>
      <meta attribute="use-in-tostring">true</meta>
```

```
</property>  
<set name="comments"table="TRACK_COMMENTS">  
.....
```

注意，我们告诉Hibernate，这个属性是UserType接口的实现类，而不是它负责持久化的原始枚举类型。因为sourceMedia属性的type配置了一个实现了UserType接口的实现类，Hibernate就会委托这个类来为sourceMedia属性执行持久化，以及查找与映射相关的Java类和SQL类型。

现在，运行ant codegen命令来更新Track类，以引入这个新的属性。

不要这么快

在开发本章示例期间，我遇到过一个奇怪的问题，代码突然不能通过编译，报告没有发现构造函数。起初，这个问题看起来好像和采用Maven Ant Tasks进行依赖管理有关，这个问题在我测试时第一次出现。仔细检查源代码，确定是不是哪出问题了。这花费了不少时间，因为这段代码很微妙。可能是因为Track类的sourceMedia属性被赋予了SourceMediaType类型的值（映射管理器），而不是它应该接受的SourceMedia类型。

在所有办法都失败以后，我将这个麻烦的bug报告给了Hibernate Tools团队，他们很快回复说不能重现那个问题，这时我明白是怎么回事

事了。构建过程之所以中断是因为：**Hibernate Tools**需要能够找到编译好的**SourceMediaType**类，映射文档才有意义，也才能够知道这个类是用户自定义的类型。在写这段文字时，我已经编写和编译好了**SourceMediaType**类，这样，当我按例6-3所示来更新映射配置并调用**codegen**构建目标时，该类编译好的类文件应该到位了。但当我回头再用**Maven Ant Tasks**进行测试时，其实这时还没有编译好的类存在，就像你在下载代码示例文件以后，再按本书章节介绍的办法来更新创建和查询测试。不过，如果在这种情况下，在**compile**之前运行**codegen**就会让你处于一种类文件不一致、不能编译的境地。但是也不能在**codegen**之前运行**compile**，因为测试类的运行要依赖于生成的数据类。

注意：听起来确实有些像经典的**catch-22**问题（因为不合逻辑的规定而造成左右为难的困境）。

很不幸，当没有很谨慎地维护构建指令时，这种令人头痛的循环依赖问题并非不常见。我为**codegen**引入了一个新的依赖，但没有在**build.xml**中定义。因为我们查找错误的方向有误，所以浪费了很多时间，但也正因为这样，我才有机会描述这一问题和它的解决方案。所以，希望当你再遇到类似的情况时，会变得更聪明些。

在清楚地理解了问题出在哪以后，就不难解决了。例6-4展示了需要对**build.xml**做的修改。

例6-4: 在构建过程中定义UserType类的依赖

```
<!--Compile the UserType definitions so they can be used in the
code
generation phase.-->
<target name="usertypes"depends="prepare"❶
description="Compile custom type definitions needed in by
codegen">
<javac srcdir="${source.root}"
includes="com/oreilly/hh/*Type.java"
destdir="${class.root}"
debug="on"
optimize="off"
deprecation="on">
<classpath refid="project.class.path"/>
</javac>
</target>
<!--Generate the java code for all mapping files in our source
tree-->
<target name="codegen"depends="usertypes"❷
description="Generate Java source from the O/R mapping files">
```

❶我们在现有的**codegen**目标之前，创建了一个名为**usertypes**的构建目标，用于编译刚才的用户类型定义。因为自定义类型不会引用任何生成的类，所以可以在**codegen**目标运行之前先编译它们。选择这些自定义类型的最简单的方法，就是利用它们位于**com.oreilly.hh**包这一事实，以及我们在这里使用的命名惯例（**Hibernate**本身也将这个包作为它保存类型映射类的地方），在这个包中它们的类文件都以**"Type.java"**结束（例如，**SourceMediaType.java**，以及本章稍后介绍的**StereoVolumeType.java**）。

如果你不喜欢这样的惯例，则可以把所有文件都列在这里，或是将它们放在其他单独的包中。这种命名方法碰巧很适合我们的需要。

❷接着，我们更新codegen构建目标，让它依赖usertypes。这样可以保证代码生成任务运行以前，它需要的自定义类型映射总能成功编译和使用（这里不需要依赖prepare，因为现在只需要依赖usertypes）。

配置好这些额外的设置以后，现在运行ant codegen就可以正确地更新Track类，以包括新的属性。完整的Track构造函数现在应该如下所示：

```
public Track (String title, String filePath, Date playTime,
Set<Artist>artists, Date added, short volume,
SourceMedia sourceMedia, Set<String>comments) {.....}
```

还需要相应地修改CreateTest.java：

```
Track track=new Track ("Russian Trance",
"vol2/album610/track02.mp3",
Time.valueOf ("00: 03: 30"),
new HashSet<Artist> () ,
new Date () , (short) 0, SourceMedia.CD,
new HashSet<String> () );
track=new Track ("Video Killed the Radio Star",
"vol2/album611/track12.mp3",
Time.valueOf ("00: 03: 49"), new HashSet<Artist> () ,
new Date () , (short) 0, SourceMedia.VHS,
new HashSet<String> () );
.....
```

为了得到如图6-1所示的结果，我们将"The World' 99"标记为来自于数字音频流，而将其他曲目都标记为来自CD，而为"Test Tone 1"标记为空的（null）sourceMedia值。这时，再运行ant schema以重建支持新属性的数据库模式，接着运行ant ctest以创建样例数据。

发生了什么事

我们的TRACK表现在包含了一个用于保存sourceMedia属性的新字段。在创建好样例数据后就可以在这个表的内容中看到它的值（最简单的方法就是用ant db命令来执行查询，结果如图6-1所示）。



The screenshot shows the HSQL Database Manager interface. On the left, a tree view displays the database structure, including the 'TRACK' table. The main window shows the SQL query 'select track_id, title, sourceMedia from track' and its results. The results are displayed in a table with three columns: TRACK_ID, TITLE, and SOURCEMEDIA. The data rows are as follows:

TRACK_ID	TITLE	SOURCEMEDIA
1	Russian Trance	CD
2	Video Killed the Radio Star	VHS
3	Gravity's Angel	CD
4	Adagio for Strings (Ferry Corsten Remix)	CD
5	Adagio for Strings (ATB Remix)	CD
6	The World '99	STREAM
7	Test Tone 1	(null)

图 6-1 TRACK表中的来源媒介信息

通过交叉检查为我们的持久化枚举类型赋值的代码，可以验证持久化保存到数据库的值是否正确。利用Java 5的enum功能，甚至可以让这种原始的查询显得更有意义。

为什么不能工作

在为我们的映射文档引入这些自定义类型的同时，也就在build.xml引入了另一种还没有反映出来过的新依赖。所以，如果一不小心，就会在运行ant shema命令之前，遇到ant编译失败的错误，收到一些Hibernate报告的以下信息：

```
[hibernatetool]INFO: Using dialect:
org.hibernate.dialect.HSQLDialect
[hibernatetool]An exception occurred while running exporter#2:
hbm2ddl
(Generates database schema)
[hibernatetool]To get the full stack trace run ant with-verbose
[hibernatetool]org.hibernate.MappingException: Could not
determine type
for: com.oreilly.hh.StereoVolumeType, for columns:
[org.hibernate.mapping
.Column (VOL_LEFT) , org.hibernate.mapping.Column (VOL_RIGHT) ]
BUILD FAILED
/Users/jim/Documents/Work/OReilly/svn_hibernate/current/examples/
ch07/build
.xml: 81: org.hibernate.MappingException: Could not determine type
for: com
.oreilly.hh.StereoVolumeType, for columns:
[org.hibernate.mapping.Column
(VOL_LEFT) , org.hibernate.mapping.Column (VOL_RIGHT) ]
Total time: 3 seconds
```

这是因为，还没有编译我们新的自定义类型，Hibernate不能发现或使用它们，所以映射起不到作用。作为一种快速修复措施，只要先运行ant compile，再试着运行ant schema就可以了。也可以在build.xml中修复这个问题，这样以后就不会再麻烦任何人了：

```
<!--Generate the schemas for all mapping files in our class
tree-->
<target name="schema"depends="compile"
description="Generate DB schema from the O/R mapping files">
.....
```

`compile`构建目标出现在`schema`后面也没有什么关系，`Ant`会安排好它们之间的依赖关系。如果你觉得不习惯，可以任意交换它们的位置。为了稳妥起见，我们也可以让`compile`依赖于`codegen`，以确保在试图编译什么以前，先生成数据类：

```
<!--Compile the java source of the project-->
<target name="compile"depends="codegen"
description="Compiles all Java classes">
.....
```

有了声明好的这些依赖，你就可以从空的源代码目录开始，一步操作就完成所有的代码生成和编译处理：

```
%ant compile
Buildfile: build.xml
prepare:
[copy]Copying 3 files to/Users/jim/svn/oreilly/hib_dev_2e/current
/examples/ch07/classes
usertypes:
[javac]Compiling 2 source files
to/Users/jim/svn/oreilly/hib_dev_2e
/current/examples/ch07/classes
codegen:
[hibernatetool]Executing Hibernate Tool with a Standard
Configuration
[hibernatetool]1.task: hbm2java (Generates a set of.java files)
compile:
[javac]Compiling 8 source files
to/Users/jim/svn/oreilly/hib_dev_2e
/current/examples/ch07/classes
BUILD SUCCESSFUL
Total time: 3 seconds
```

好，我们回头再继续学习自定义类型吧。

为了能够看到更友好的提示信息（顺便也测试一下自定义持久化辅助工具的部分检索功能），我们可以扩充一下查询测试，让它为检索回的曲目打印输出这个属性关联的描述。必要的修改如例6-5中粗体字部分所示。

例6-5: 在QueryTest.java中显示来源媒介信息

```
.....
//Print the tracks that will fit in seven minutes
List tracks=tracksNoLongerThan (Time.valueOf ("00: 07: 00") ,
session) ;
for (ListIterator iter=tracks.listIterator () ;
iter.hasNext () ; ) {
Track aTrack= (Track) iter.next () ;
String mediaInfo="";
if (aTrack.getSourceMedia () !=null) {
mediaInfo=", from"+
aTrack.getSourceMedia () .getDescription () ;
}
System.out.println ("Track: \""+aTrack.getTitle () +"\""+
listArtistNames (aTrack.getArtists () ) +
aTrack.getPlayTime () +mediaInfo) ;
}
.....
```

增加了这些扩充的代码后，运行ant qtest，其输出结果如例6-6所示。具有非空（non-null）来源媒介值的曲目后面会跟着一个"from"，之后显示的就是该曲目相应的媒介描述信息。

例6-6: 方便阅读的来源媒介信息显示

```
.....
qtest:
[java]Track: "Russian Trance" (PPK) 00: 03: 30, from Compact Disc
```

```
49, [java]Track: "Video Killed the Radio Star" (The Buggles) 00: 03:
    from VHS Videocassette tape
    [java]Track: "Gravity's Angel" (Laurie Anderson) 00: 06: 06, from
    Compact Disc
    [java]Track: "Adagio for Strings (Ferry Corsten Remix) " (William
    Orbit, Ferry Corsten, Samuel Barber) 00: 06: 35, from Compact Disc
    [java]Track: "Test Tone 1"00: 00: 10
    [java]Comment: Pink noise to test equalization
```

注意，如果我们决定在`QueryTest`中不自己处理曲目属性子集的格式化输出，而是要依靠`Track`类的`toString()`方法，那么我们不需要对`QueryTest`进行任何修改就可以看到这种新的输出信息，虽然用数据库查询我们已经看到了同样的枚举名称列表的最简单版本。我们在映射文档中规定`toString()`方法返回的结果应该包含`sourceMedia`属性值，该方法负责处理这个属性值。可以检查生成的`toString()`方法的源代码以验证这一点，或者编写一段简单的程序来看看`toString()`方法输出的结果是什么样的。当然，一种很好的策略就是修改`AlbumTest.java`，让它在我们修改`Track`以后可以通过编译并运行。最简单的修改就是在`addAlbumTrack()`方法中通过硬编码（hard-code）让每个曲目都来自CD，如例6-7所示（`JavaDoc`已经为这种只图简单的做法想好了理由）。

例6-7：修改`AlbumTest.java`，增加对曲目来源媒介的支持

```
/**
 *Quick and dirty helper method to handle repetitive portion of
 creating
 *album tracks.A real implementation would have much more
 flexibility.
```

```

    */
    private static void addAlbumTrack (Album album, String title,
String file,
    Time length, Artist artist, int disc,
    int positionOnDisc, Session session) {
        Track track=new Track (title, file, length, new HashSet<Artist>
    ( ) ,
        new Date ( ) , (short) 0, SourceMedia.CD,
        new HashSet<String> ( ) ) ;
        track.getArtists ( ) .add (artist) ;
        //session.save (track) ;
        album.getTracks ( ) .add (new AlbumTrack (track, disc,
positionOnDisc) ) ;
    }

```

修改好文件以后，运行ant atest命令，就会显示在Album的toString
() 方法中生成的来源媒介信息：

```

[java]com.oreilly.hh.data.Album@ccad9c[title='Counterfeit
e.p.'tracks='[
    com.oreilly.hh.data.AlbumTrack@9c0287[track='com.oreilly.hh.data.
Track@6a21b2[
    title='Compulsion'sourceMedia='CD']'],
com.oreilly.hh.data.AlbumTrack@aa8eb7
    [track='com.oreilly.hh.data.Track@7fc8a0[title='In a Manner of
Speaking'source
    Media='CD']'],
com.oreilly.hh.data.AlbumTrack@4cad4c[track='com.oreilly.hh.da
ta.Track@243618[title='Smile in the Crowd'sourceMedia='CD']'],
com.oreilly.h
    h.data.AlbumTrack@5b644b[track='com.oreilly.hh.data.Track@157e43[
title='Gone'
    sourceMedia='CD']'],
com.oreilly.hh.data.AlbumTrack@1483a0[track='com.oreilly
    .hh.data.Track@cdae24[title='Never Turn Your Back on Mother
Earth'sourceMedia=
    'CD']'],
com.oreilly.hh.data.AlbumTrack@63dc28[track='com.oreilly.hh.data.Tra
ck@ae511[title='Motherless Child'sourceMedia='CD']']']]']

```

没有做多少工作，我们就利用**Hibernate**扩展了能够支持持久化的类型安全的枚举类。在付出一定的努力之后，就可以像**Hibernate**支持的其他原生值类型一样，方便地对这些枚举类进行持久化。

如果以后不用写代码，**Hibernate**中支持的原生类型可以直接利用Java 5支持的**enum**关键字，那就太好了。不过，对此我并不抱多少希望，因为Java 5问世已经有一段时间了。但是，就各种评论来说，这是一种“温和”（**mild**）的方法，**Hibernate wiki**上还有其他可供选择的支持枚举类型的用户自定义类型的实现。

接下来，我们将介绍如何映射更加复杂和特殊的自定义类型，没有人会指望**Hibernate**能够为这么复杂的类型映射提供内建的支持。

建立组合自定义类型

回想一下，我们的Track对象中有个属性，用于决定曲目在播放时音量的大小。假设我们希望自动唱机系统除了能控制曲目的音量以外，也能够调整播放曲目时音质的均衡度（balance）。为了实现这一点，我们需要为左右声道分别保存音量。一种快速的解决方案就是编辑Track映射文档，将这些功能要求映射到单独的属性。

如果我们严格地从面向对象架构的角度来考虑，可能希望将这两个音量值封装到一个StereoVolume类中。然后，再将这个类直接映射到一个composite-element元素，就像我们在例5-4中对AlbumTrack组件所做的那样。这依然相当直截了当。

不过，这种简单的解决方案有个缺点。系统中的其他地方可能也需要StereoVolume值。如果我们建立一种播放列表机制，它可以改写曲目默认的播放选项，同时又能对整个专辑的播放音量进行控制。突然间，我们就得在好几个地方重建组合映射配置，而且有可能无法保持前后一致（对于更复杂的复合类型来说，这更可能是个问题，但现在只需要有个想法就可以了）。Hibernate参考文档指出，像这种情况，使用一个自定义的复合类（composite user type）会是种很好的务实做法，我也同意这一点。

应该怎么做

我们先从定义StereoVolume类开始做起。没有理由要将这个类作为实体（使其独立存在于其他持久化对象以外），所以只要将它作为普通的（而且相当简单的）Java对象来实现即可。例6-8展示了它的代码。

注意：该例的JavaDoc经过精简以节省空间。相信你在实际项目中不会这么做。从网站下载的版本则更加完整。

例6-8：一个用于表示音量等级的值类（StereoVolume.java）

```
package com.oreilly.hh;
import java.io.Serializable;
/**
 *A simple structure encapsulating a stereo volume level.
 */
public class StereoVolume implements Serializable{
    /**The minimum legal volume level.*/
    public static final short MINIMUM=0;
    /**The maximum legal volume level.*/
    public static final short MAXIMUM=100;
    /**Stores the volume of the left channel.*/
    private short left;
    /**Stores the volume of the right channel.*/
    private short right;
    /**Default constructor sets full volume in both channels.*/
    public StereoVolume () {❶
        this (MAXIMUM, MAXIMUM) ;
    }
    /**Constructor that establishes specific volume levels.*/
    public StereoVolume (short left, short right) {
        setLeft (left) ;
        setRight (right) ;
    }
    /**
```

```

*Helper method to make sure a volume value is legal.
@param volume the level that is being set.
*throws IllegalArgumentException if it is out of range.
*/
private void checkVolume (short volume) {
    if (volume<MINIMUM) {
        throw new IllegalArgumentException ("volume cannot be less than"+
            MINIMUM) ;
    }
    if (volume>MAXIMUM) {
        throw new IllegalArgumentException ("volume cannot be more than"+
            MAXIMUM) ;
    }
}
/**Set the volume of the left channel.*/
public void setLeft (short volume) {❷
    checkVolume (volume) ;
    left=volume;
}
/**Set the volume of the right channel.*/
public void setRight (short volume) {
    checkVolume (volume) ;
    right=volume;
}
/**Get the volume of the left channel*/
public short getLeft () {
    return left; }
/**Get the volume of the right channel.*/
public short getRight () {
    return right; }
/**Format a readable version of the volume levels, for
debugging.*/
public String toString () {
    return"Volume[left="+left+", right="+right+"]";
}
/**
*Compare whether another object is equal to this one.
@param obj the object to be compared.
*return true if obj is also a StereoVolume instance, and
represents
*the same volume levels.
*/
public boolean equals (Object obj) {❸
    if (obj instanceof StereoVolume) {
        StereoVolume other= (StereoVolume) obj;
        return other.getLeft () ==getLeft () &&
            other.getRight () ==getRight () ;
    }
}

```

```
        return false; //It wasn't a StereoVolume
    }
    /**
    *Returns a hash code value for the StereoVolume.This method must
be
    *consistent with the{@link#equals}method.
    */
    public int hashCode () {
        return (int) getLeft () *MAXIMUM*10+getRight () ;
    }
}
```

❶ 因为我们想用Hibernate持久化这个对象，所以必须提供一个默认的构造函数。

❷ 以及属性访问器。

❸ 提供对Java equals () 和hashCode () 比较的支持也是重要的，因为这是一个可变值对象。

为了将这个类作为复合类型进行持久化保存，而非每次使用时都将其定义为嵌套的组合对象，我们需要建立一个复合自定义类型来管理它的持久化处理。在该自定义类型中所需要提供的大部分处理和我们在SourceMediaType（例6-2，本章前面演示的一个例子）中提供的差不多。所以这里只集中介绍新鲜而有趣的内容。例6-9演示了以复合自定义类型的方式来持久化StereoVolume类的一种方法。

例6-9：用于持久化StereoVolume的复合自定义类型
(StereoVolumeType.java)

```

package com.oreilly.hh;
import java.io.Serializable;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.hibernate.Hibernate;
import org.hibernate.engine.SessionImplementor;
import org.hibernate.type.Type;
import org.hibernate.usertype.CompositeUserType;
/**
 *Manages persistence for the{@link StereoVolume}composite type.
 */
public class StereoVolumeType implements CompositeUserType{
/**
 *Get the names of the properties that make up this composite
type, and
 *that may be used in a query involving it.
 */
public String[]getPropertyNames () {❶
//Allocate a new response each time, because arrays are mutable
return new String[]{"left", "right"};
}
/**
 *Get the types associated with the properties that make up this
composite
 *type.
 *
 *@return the types of the parameters reported
by{@link#getPropertyNames},
 *
in the same order.
 */
public Type[]getPropertyTypes () {
return new Type[]{Hibernate.SHORT, Hibernate.SHORT};
}
/**
 *Look up the value of one of the properties making up this
composite type.
 *
 *@param component a{@link StereoVolume}instance being managed.
 *@param property the index of the desired property.
 *@return the corresponding value.
 *@see#getPropertyNames
 */
public Object getPropertyValue (Object component, int property)
{❷
StereoVolume volume= (StereoVolume) component;
short result;

```

```

        switch (property) {
        case 0:
            result=volume.getLeft () ;
            break;
        case 1:
            result=volume.getRight () ;
            break;
        default:
            throw new IllegalArgumentException ("unknown
property: "+property) ;
        }
        return new Short (result) ;
    }
    /**
     *Set the value of one of the properties making up this composite
type.
     *
     * @param component a{@link StereoVolume}instance being managed.
     * @param property the index of the desired property.
     * @param value the new value to be established.
     * @see#getPropertyNames
     */
    public void setPropertyValue (Object component, int property,
Object value) {
        StereoVolume volume= (StereoVolume) component;
        short newLevel= ( (Short) value) .shortValue () ;
        switch (property) {
        case 0:
            volume.setLeft (newLevel) ;
            break;
        case 1:
            volume.setRight (newLevel) ;
            break;
        default:
            throw new IllegalArgumentException ("unknown
property: "+property) ;
        }
    }
    /**
     *Determine the class that is returned by{@link#nullSafeGet}.
     *
     * @return{@link StereoVolume}, the actual type returned by
     *
     * {@link#nullSafeGet}.
     */
    public Class returnedClass () {
        return StereoVolume.class;
    }
}

```

```

/**
 *Compare two instances of the class mapped by this type for
persistence
 *"equality".
 *
 *@param x first object to be compared.
 *@param y second object to be compared.
 *@return<code>true</code> iff both represent the same volume
levels.
 *@throws ClassCastException if x or y isn't a{@link
StereoVolume}.
 */
public boolean equals (Object x, Object y) {③
if (x==y) {//This is a trivial success
return true;
}
if (x==null||y==null) {//Don't blow up if either is null!
return false;
}
//Now it's safe to delegate to the class'own sense of equality
return ( (StereoVolume) x) .equals (y) ;
}
/**
 *Return a deep copy of the persistent state, stopping at entities
and
 *collections.
 *
 *@param value the object whose state is to be copied.
 *@return a copy representing the same volume levels as the
original.
 *@throws ClassCastException for non{@link StereoVolume}values.
 */
public Object deepCopy (Object value) {④
if (value==null)
return null;
StereoVolume volume= (StereoVolume) value;
return new StereoVolume (volume.getLeft () , volume.getRight
() ) ;
}
/**
 *Indicates whether objects managed by this type are mutable.
 *
 *@return<code>true</code>, since{@link StereoVolume}is
mutable.
 */
public boolean isMutable () {
return true;
}

```

```

/**
 *Retrieve an instance of the mapped class from a JDBC{@link
ResultSet}.
 *
 *{@param rs the results from which the instance should be
retrieved.
 *{@param names the columns from which the instance should be
retrieved.
 *{@param session an extension of the normal Hibernate session
interface
 *that gives you much more access to the internals.
 *{@param owner the entity containing the value being retrieved.
 *{@return the retrieved{@link StereoVolume}value, or<code>null
</code>.
 *{@throws SQLException if there is a problem accessing the
database.
 */
public Object nullSafeGet (ResultSet rs, String[]names, ⑤
SessionImplementor session, Object owner)
throws SQLException{
Short left= (Short) Hibernate.SHORT.nullSafeGet (rs, names[0]) ;
Short right= (Short) Hibernate.SHORT.nullSafeGet (rs, names[1]) ;
if (left==null||right==null) {
return null; //We don't have a specified volume for the channels
}
return new StereoVolume (left.shortValue () , right.shortValue
() ) ;
}
/**
 *Write an instance of the mapped class to a{@link
PreparedStatement},
 *handling null values.
 *
 *{@param st a JDBC prepared statement.
 *{@param value the StereoVolume value to write.
 *{@param index the parameter index within the prepared statement
at which
 *this value is to be written.
 *{@param session an extension of the normal Hibernate session
interface
 *that gives you much more access to the internals.
 *{@throws SQLException if there is a problem accessing the
database.
 */
public void nullSafeSet (PreparedStatement st, Object value, int
index,
SessionImplementor session)
throws SQLException{

```



```

        if (value==null) {
            Hibernate.SHORT.nullSafeSet (st, null, index) ;
            Hibernate.SHORT.nullSafeSet (st, null, index+1) ;
        }else{
            StereoVolume vol= (StereoVolume) value;
            Hibernate.SHORT.nullSafeSet (st, new Short (vol.getLeft () ) ,
index) ;
            Hibernate.SHORT.nullSafeSet (st, new Short (vol.getRight () ) ,
index+1) ;
        }
    }
}
/**
 *Reconstitute a working instance of the managed class from the
cache.
 *
 * @param cached the serializable version that was in the cache.
 * @param session an extension of the normal Hibernate session
interface
 *that gives you much more access to the internals.
 * @param owner the entity containing the value being retrieved.
 * @return a copy of the value as a{@link StereoVolume}instance.
 */
    public Object assemble (Serializable cached, SessionImplementor
session, ⑥
    Object owner) {
        //Our value type happens to be serializable, so we have an easy
out.
        return deepCopy (cached) ;
    }
}
/**
 *Translate an instance of the managed class into a serializable
form to be
 *stored in the cache.
 *
 * @param session an extension of the normal Hibernate session
interface
 *that gives you much more access to the internals.
 * @param value the StereoVolume value to be cached.
 * @return a serializable copy of the value.
 */
    public Serializable disassemble (Object value, SessionImplementor
session) {
        return (Serializable) deepCopy (value) ;
    }
}
/**
 *Get a hashCode for the instance, consistent with
persistence"equality"
 */

```

```

    public int hashCode (Object x) {❶
    return x.hashCode () ; //Can delegate to our well-behaved object
    }
    /**
    *During merge, replace the existing (target) value in the entity
we are
    *merging to with a new (original) value from the detached entity
we are
    *merging.For immutable objects, or null values, it is safe to
simply
    *return the first parameter.For mutable objects, it is safe to
return a
    *copy of the first parameter.However, since composite user types
often
    *define component values, it might make sense to recursively
replace
    *component values in the target object.
    *
    *@param original value being merged from.
    *@param target value being merged to.
    *@param session the hibernate session into which the merge is
happening.
    *@param owner the containing entity.
    *@return an independent value that can safely be used in the new
context.
    */
    public Object replace (Object original, Object target, ❷
    SessionImplementor session, Object owner) {
    return deepCopy (original) ;
    }
    }

```

❶由于有了`getPropertyNames ()`和`getPropertyTypes ()`方法，**Hibernate**就可以知道组成复合类型的各“组成部分”。当编写HQL查询时就可以使用这些值类型。在这个例子中它们相当于我们正在持久化的实际**StereoVolume**类的属性值，但不是必需的。例如，我们可以借这个机会来为根本不是为持久化而设计的遗留（**legacy**）类提供一个友好的属性接口。

❷复合用户定义类型的虚拟属性和它们基于的真实数据之间转换是由`getPropertyValue ()`和`setPropertyValue ()`方法提供的。在本质上，**Hibernate**只是给了我们一个想要管理的类型的实例，而且没有一点假设，它只是说“嗨，给我第二个属性”，或者是说“把第一个属性设置为这个值”。你可以看到如何用这种方法为旧的或第三方代码增加属性接口。在这个例子中，因为我们实际上不需要这种功能，我们要将属性处理传递给底层的**StereoVolume**类，这里要跨越的障碍只是模板类。

接下来的一大段代码由前面例6-2中见过的方法组成，不过例子中的版本具有的一些区别很有意思。大部分变化与前面提到的内容有关，不像**SourceMedia**，我们的**StereoVolume**类是可变的，它包含了可变化的值。所以，我们得为一些最终适合的方法设计出完整的实现。

❸我们需要在`equals ()`中提供一种有意义的方法来比较实例。

❹还要在`deepCopy ()`中实现独立的实例复制。

❺实际的持久化方法（`nullSafeGet ()`和`nullSafeSet ()`）与例6-2非常相似，只有一点不同，我们对此不需要过多介绍。它们都有一个**SessionImplementor**参数，通过这个参数可以访问让**Hibernate**正常工作的内部组件。只有真正复杂的持久化处理才需要使用这个参数，这已经超出本书介绍的范围。如果你需要使用**SessionImplementor**方法，在

实现上相当有技巧，必须深刻理解Hibernate的体系结构。其实你正写的是对系统的扩展，可能需要研究源代码才能获得必需的专业知识。

⑥ `assemble ()` 和 `disassemble ()` 方法可以让自定义类型支持对非 `Serializable` 值的缓存。它们让我们的持久化工具方法有机会可以将任何重要的值复制到另一个能够被序列化 (`serialized`) 的对象中 (使用任何必要的手段)。因为让 `StereoVolume` 首先成为可序列化的并不重要，我们也不需要这种灵活性，所以我们的实现只是为了能保存在缓存中复制一些可序列化的 `StereoVolume` 实例 (之所以要复制实例，是因为我们的数据类是可变的，不应该让缓存值也莫名其妙地发生变化)。

⑦ `hashCode ()` 方法是Hibernate 3中新增加的方法，虽然需要修改 `CompositeUserType` 实现，但它有助于提高效率。在这个例子中，我们有一个对象已经实现了这个方法，可以将处理委托给这个对象。但是，再一次强调，如果我们正在封装一些糟糕的遗留数据结构，就可以借这个机会为它们增加一个漂亮的Java包装器。

⑧ 最后，`replace ()` 方法是Hibernate 3要求的另一个新方法。再一次，因为我们需要复制一个对象，可以用一种容易的办法来实现。另外，我们也可以手工将所有内嵌的属性值从最初的对象复制到目标对象。

注意：这么一个简单的值类居然需要做这么多的工作。但是，这正是对更复杂的值类进行建模的好起点。

好了，这头“野兽”已经创建完成，接下来应该怎么用？为了使用该新的复合类型，例6-10改进了Track映射文档中volume属性的配置，同时为了便于查看测试的输出，也趁此机会将它加到了toString () 方法中。

例6-10：修改Track.hbm.xml以使用StereoVolume

```
.....
    <property name="volume" type="com.oreilly.hh.StereoVolumeType">
      <meta attribute="field-description">How loud to play the track
</meta>
      <meta attribute="use-in-tostring">true</meta>
      <column name="VOL_LEFT"/>
      <column name="VOL_RIGHT"/>
    </property>
.....
```

再次注意，映射文档中提供的是负责管理持久化的自定义类型的名称，而不是它所管理的原始类型。这与例6-3是一样的。此外，这个复合类型使用两个字段存储数据，所以这里也得提供两个字段名称。

现在，当我们执行ant codegen命令，为Track重新生成Java源代码时，可以得到例6-11所示的结果。

例6-11：新生成的Track.java源代码的变动之处

```

.....
/**
 *How loud to play the track
 */
private StereoVolume volume;
.....
public Track (String title, String filePath, Date playTime,
Set<Artist>artists, Date added,
StereoVolume volume, SourceMedia sourceMedia,
Set<String>comments) {
.....
}
.....
/**
 *How loud to play the track
 */
public StereoVolume getVolume () {
return this.volume;
}
public void setVolume (StereoVolume volume) {
this.volume=volume;
}
.....
public String toString () {
StringBuffer buffer=new StringBuffer () ;
buffer.append (getClass () .getName () ) .append ("@" ) .append
(Integer.toHexString
(hashCode () ) ) .append ("[" ) ;
buffer.append ("title" ) .append ("=" ) .append (getTitle
() ) .append ("'" ) ;
buffer.append ("volume" ) .append ("=" ) .append (getVolume
() ) .append ("'" ) ;
buffer.append ("sourceMedia" ) .append ("=" ) .append
(getSourceMedia () ) .append (
"'" ) ;
buffer.append ("]" ) ;
return buffer.toString () ;
}
.....

```

此时，可以执行ant schema命令来重建数据库表，例6-12演示了相关的结果。

例6-12: 根据新的映射而创建的曲目数据库模式

```
.....
[hibernatetool]create table TRACK (TRACK_ID integer generated by
default as
identity (start with 1) , TITLE varchar (255) not null,
filePath varchar (255) not null, playTime time, added date,
VOL_LEFT smallint, VOL_RIGHT smallint, sourceMedia varchar
(255) ,
primary key (TRACK_ID) ) ;
.....
```

让我们进一步改进数据创建的测试程序, 使其能够采用新的Track结构。例6-13演示了我们需要做出的修改。

例6-13: 对CreateTest.java做必要的修改以测试立体声音量

```
.....
//Create some data and persist it
tx=session.beginTransaction () ;
StereoVolume fullVolume=new StereoVolume () ;
Track track=new Track ("Russian Trance",
"vol2/album610/track02.mp3",
Time.valueOf ("00: 03: 30") ,
new HashSet<Artist> () ,
new Date () , fullVolume, SourceMedia.CD,
new HashSet<String> () ) ;
addTrackArtist (track, getArtist ("PPK", true, session) ) ;
session.save (track) ;
.....
//The other tracks created use fullVolume too, until.....
.....
track=new Track ("Test Tone 1",
"vol2/singles/test01.mp3",
Time.valueOf ("00: 00: 10") , new HashSet<Artist> () ,
new Date () , new StereoVolume ( (short) 50, (short) 75) ,
null, new HashSet<String> () ) ;
track.getComments () .add ("Pink noise to test equalization") ;
session.save (track) ;
.....
```

现在，如果我们执行ant ctest，并用ant db查看结果，就会看到如图6-2所示的结果。

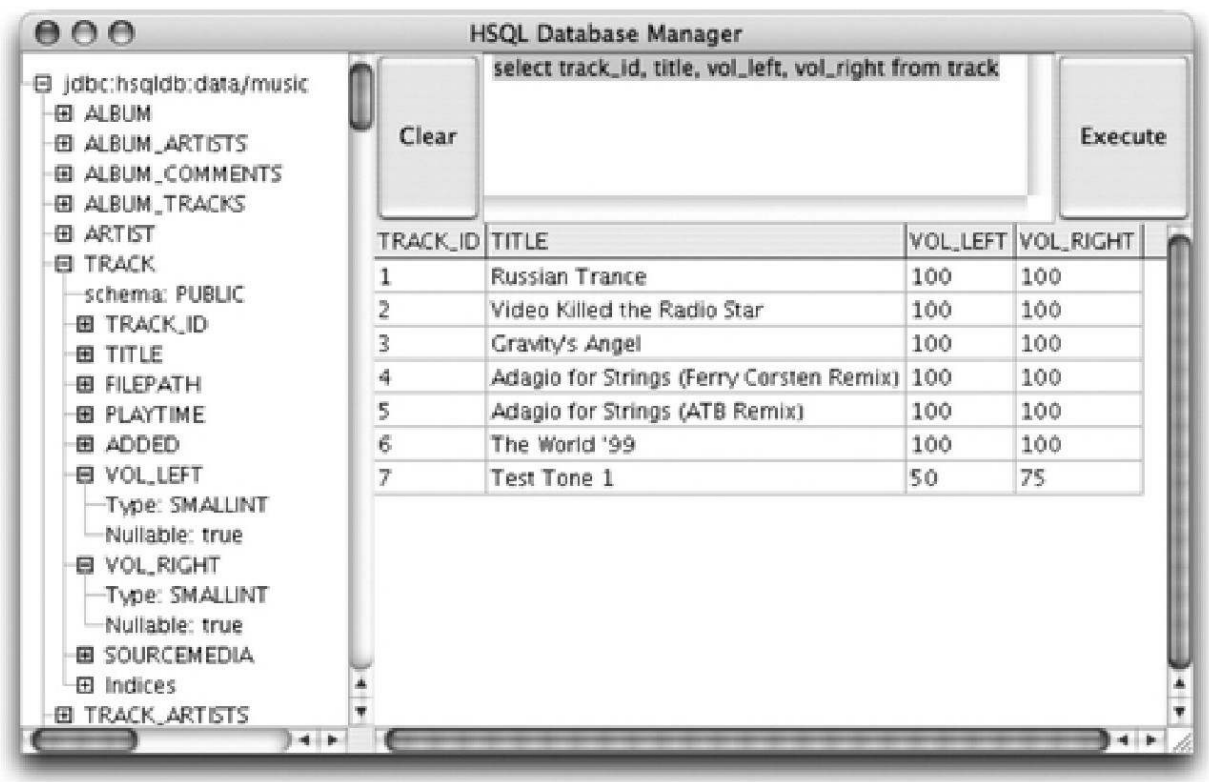


图 6-2 TRACK表中的立体声音量信息

为了让AlbumTest与新的Track格式保持兼容，我们只需要对AlbumTest做一处修改，如例6-14所示。

例6-14: 对AlbumTest.java所做的修改，以支持立体声曲目音量

```
.....
private static void addAlbumTrack (Album album, String title,
String file,
Time length, Artist artist, int disc,
int positionOnDisc, Session session) {
Track track=new Track (title, file, length, new HashSet<Artist>
() ,
.....
.....
}
```



```
new Date () , new StereoVolume () , SourceMedia.CD,  
new HashSet<String> () ) ;  
.....
```

这样我们就可以运行`ant atest`命令，查看新版的Track的`toString ()`方法所显示的音量信息，如例6-15所示。

例6-15：带有立体声音量信息的专辑

```
[java]com.oreilly.hh.data.Album@ccad9c[title='Counterfeit  
e.p.'tracks='[com.oreilly.hh.data.AlbumTrack@9c0287[track='com.oreil  
ly.hh.data.Track@6a21b2[title='Compulsion'volume='Volume[left=100,  
right=100]'sourceMedia='CD']'], c  
om.oreilly.hh.data.AlbumTrack@aa8eb7[track='com.oreilly.hh.data.Trac  
k@7fc8a0[title='In a Manner of Speaking'volume='Volume[left=100,  
right=100]'sourceMedia='CD']'],  
com.oreilly.hh.data.AlbumTrack@4cad4c[track='com.oreilly.hh.data.Tra  
ck@243618[title='Smile in the Crowd'volume='Volume[left=100,  
right=100]'sourceMedia='CD']'],  
com.oreilly.hh.data.AlbumTrack@5b644b[track='com.oreilly.hh.d  
ata.Track@157e43[title='Gone'volume='Volume[left=100,  
right=100]'sourceMedia='CD']'],  
com.oreilly.hh.data.AlbumTrack@1483a0[track='com.oreilly.hh.data.Tra  
ck@cdae24[title='Never Turn Your Back on Mother  
Earth'volume='Volume[left=100, right=100]'sourceMedia='CD']'],  
com.oreilly.hh.data.AlbumTrack@63dc28[track='com.oreilly.hh.data.Tra  
ck@ae511[title='Motherless Child'volume='Volume[left=100,  
right=100]'sourceMedia='CD']]']']']
```

嗯，这里介绍的内容可能深入了一点，超过了你目前对自定义类型所需要的程度。但是，总有一天你也许会回过头来深入研究这个例子，找到你正在寻找的主题。同时，接下来让我们换档去看一看完全不同、全新而且简单的事物。第7章将介绍一种完全取代XML映射文档

的方法。第8章将介绍条件查询，这是Hibernate独特的功能，对于程序员来说这个功能非常友好。

其他

映射自定义类型还有哪些奇特的诀窍？好吧，如果本章介绍的信息还不够用，你可以研究一下`org.hibernate.usertype`包中的其他接口，包括`EnhancedUserType`（这个接口将自定义类型作为实体的id，以及其他有趣的窍门）和`ParameterizedUserType`（可以配置它支持多种不同类型的映射）等。在Hibernate维基的Java 5 EnumUserType（[\[1\]](http://www.hibernate.org/272.html)）页面上也讨论了针对Java 5 enum枚举类型的可重用映射，这些都很好地演示了各种用法，只是对它们的讨论已经超出了本书的范围。

[\[1\]](http://www.hibernate.org/272.html) <http://www.hibernate.org/272.html>.

第7章 映射标注

到目前为止，我们一直在用**XML**映射文档作为我们示例的起点。在这种情况下，以一个概念模型作为开始，将创建数据表和数据对象的细节都留给**Hibernate**，同时也保留很多可供选择的余地。然而，**Java 5**对标注（**Annotation**）机制提供了灵活的支持，这一技术的出现为**Hibernate**映射提供了一种非常有趣的替代方法。尤其是当你已经拥有了一些现有的对象，只是在考虑怎么将它们保存到数据库时，就是这种新的替代方法的适用场合了。

Hibernate标注

如果你以前从没有接触过标注，本章的代码示例将看起来有点奇怪，所以，现在最好先花点时间讨论一下**Java**标注的历史和目的。基本上，一个标注就是为一段代码添加一些信息（在**Java**世界中，通常是一个类、字段或方法），以便帮助其他工具理解如何使用这段代码，或是进行某些自动化处理以节约你的工作量。与持久化映射不同的是，为了与源代码保持一致，不用将标注放到一个单独的文件中，而是直接将标注放在它所影响的源代码文件中。采用这种方法，你就不用担心与源代码分离保存的文件是否会变得与源代码不同步了。早

在Java 5为这种编码风格提供健壮的支持以前，人们就通过利用JavaDoc工具可扩展的特性，发现了一种取得这种支持的“后门”。

JavaDoc也是一种形式的标注，从Java开始就一直存在，它的目的是让开发人员为他们的类和API生成高质量的文档，而且还不必维护除源代码以外的任何文件。JavaDoc的工作效果非常好，所以人们就想用它来做些其他事情。XDoclet ([\[1\]](#)) 项目是一个很流行，而且也比较复杂的框架，它以多种有趣的方式对JavaDoc进行了扩展。Hibernate也开发了一套丰富的Hibernate XDoclet标签。

也正因为这种方法的功能如此强大，所以Sun在Java 5中内建了完整的、通用的标注支持。这样，就不再需要那些借用JavaDoc注释的复杂工具了。现在Java编译器本身就可以处理标注（并提供了它自己的一些标注来控制单独的类、方法甚至字段上的特定警告信息）。利用反射（**reflection**）机制来解析标注（如果将它们配置为要保留在编译后的类中），也能够非常容易地定义自己的标注类。所以，Hibernate自然采用了Java 5的原生（**native**）标注。

趋同进化（**convergent evolution**）的速度如此之快，使用标注为类配置映射的能力具有的强大吸引力，已经让Hibernate自己的标签深深地影响了EJB 3规范。如此有用的Hibernate风格的持久化要是能够在成熟的Java Enterprise Edition（EE）环境以外也可以使用，这给人们留下深刻的印象，所以他们定义了Java Persistence API（经常称为

JPA），作为一种可以在普通的Java Standard Edition（SE）环境可以独立使用的持久化组件。因为有规范作为基础，Hibernate自然会采用直接支持它们，所以现在通过EJB 3和JPA接口以及标注，也可以使用Hibernate能够提供的许多功能，而应用程序本身根本不必依赖（或知道）Hibernate。

我们不打算介绍那么多功能，因为我们在Hibernate中使用的一些功能需要使用它自己的标签。事实上，在你习惯使用Hibernate，并领会到它是一种非常灵活而强大的体系，可以“按照你需要的任何方式，持久化任何普通Java对象”以后，你会发现按照JPA规范的要求来封装你的代码将不得不考虑太多的限制，除非项目确实强制要求需要屏蔽底层的持久化实现（可能你会想办法尽量避免这样的项目）。不过，如果你正在使用Hibernate，有时使用标注而不是XML文件来配置映射，也是一种不错的选择。

为何在意

在熟悉了标注的语法以后（一些像Eclipse之类的IDE已经提供了惊人的能力，支持标注的理解、文档化以及自动完成，甚至自定义供你自己使用的标注，第14章将介绍这种技术），编写映射文档就只需要涉及一种文件格式的建立和读取了。

注意：什么！为什么不早点介绍这么酷的东西？

如前所述，如果将数据对象和映射规定都放在同一个文件中，那么它们不能保持同步的可能性就很小，阅读代码也可以马上理解正在进行的处理。添加标注后的代码将是一种自文档化的（**self-documenting**）代码。一些标注提供的默认设置经常也可以减少你需要进行配置的工作量。用标注指定所有设置，你会发现标注语法比XML文件更精简，而且需要输入的文字量比较少，或者让IDE帮你完成输入后，只要读一下就可以了。

当你可以直接控制数据库时，这种方法可能会很有成效，只是这种方法会将映射和数据对象紧密地耦合在一起（或者，如果你正在设计数据对象以处理特定的、固定格式的数据库结构，这样也同样会产生紧密耦合）。这意味着，如果你想用一个对象模型来支持多个数据库，使用标注就不是一种好的选择了；在这种情况下，外部映射文件的独立性其实是它的优势，因为不需要修改Java源代码，只要为不同的数据库提供单独的映射文件就可以了。

许多其他Java工具也采用标注作为配置和集成的手段，第13章将介绍相关内容。在使用这些工具时，同时也使用Hibernate的标注也是非常方便的。其实，其他工具可能也要依赖Hibernate标注。就算我们不计划在本书的新版中介绍标注的使用，介绍Spring的那一章也会强制我们面对这一问题。

对于标注的批评之一就是它让数据库细节分散到Java源代码的各个角落。所幸，一些有用的Hibernate工具可以根据Hibernate XML映射文件和Hibernate标注生成Hibernate映射文档。有关hbm2doc的更多信息，请参阅第12章的12.8.2节。

应该怎么做

我们需要做的第一件事就是更新Maven的依赖配置，让它取回Hibernate标注库。编辑build.xml文件中的依赖设置，如例7-1所示（增加的部分用粗体显示）。

例7-1：获得Hibernate标注

```
<artifact: dependencies pathId="dependency.class.path">
  <dependency
groupId="hsqldb"artifactId="hsqldb"version="1.8.0.7"/>
  <dependency groupId="org.hibernate"artifactId="hibernate"
version="3.2.5.ga">
    <exclusion groupId="javax.transaction"artifactId="jta"/>
  </dependency>
  <dependency groupId="org.hibernate"artifactId="hibernate-tools"
version="3.2.0.beta9a"/>
  <dependency groupId="org.hibernate"artifactId="hibernate-
annotations"
version="3.3.0.ga"/>
  <dependency groupId="org.hibernate"
artifactId="hibernate-commons-annotations"
version="3.3.0.ga"/>
  <dependency groupId="org.apache.geronimo.specs"
artifactId="geronimo-jta_1.1_spec"version="1.1"/>
  <dependency groupId="log4j"artifactId="log4j"version="1.2.14"/>
</artifact: dependencies>
```

在这个文件中，删除usertypes和codegen两个构建目标。因为我们要使用标注，所以就不用生成Java代码了。另一方面，我们将从Java代码文件入手，用它来定义Hibernate映射（以及数据库模式），所以在本章中应该废弃这两个构建目标（它们甚至是危险的）。

可以想到，为了适应这种新方法，还需要对schema构建目标稍微调整一下，如例7-2中突出显示部分所示。

例7-2：使用标注生成数据库模式

```
<!--Generate the schemas for annotated classes-->❶
<target name="schema"depends="compile"
description="Generate DB schema from the annotated model
classes">
  <hibernatetool destdir="${source.root}">
    <classpath refid="project.class.path"/>❷
    <annotationconfiguration
configurationfile="${source.root}/hibernate.cfg.xml"/>❸
    <hbm2ddl drop="yes"/>
  </hibernatetool>
</target>
```

❶需要更新注释和构建目标描述，以反映新的工作方式。

❷我们需要引用编译好的类，以便模式生成工具可以找到其中的标注。注意，这意味着在生成模式之前，需要先把类编译好，这样的处理顺序与该构建目标前面的大多数版本都不相同，但是在第6章中增加了这个依赖，所以我们的schema构建目标已经依赖编译目标了。

❸最后，让工具使用标注来配置它自己。我们仍旧提供一个全局的Hibernate配置文件，以便工具可以知道我们正在使用什么数据库等信息。这个文件也必须列出我们想使用的所有标注类，在调整完所有构建文件后，我们再处理它。

我们的compile构建目标需要依赖刚才删除掉的代码生成构建目标，所以在删除这一依赖以前，这个构建目标不能运行。在我们的新方法中，为了支持编译而所有需要做的就是让基本的prepare构建目标可以运行。编辑compile构建目标，以反映例7-3中突出显示部分的变化。

例7-3: 更简单的编译依赖

```
<!--Compile the java source of the project-->
<target name="compile"depends="prepare"
description="Compiles all Java classes">
  <javac srcdir="${source.root}"
  destdir="${class.root}"
  debug="on"optimize="off"deprecation="on">
  <classpath refid="project.class.path"/>
</javac>
</target>
```

如前面所述，我们需要更新Hibernate配置，将原来使用的映射文档列表替换为相应标注过的类（annotated classes）的列表。删除那些映射文档，修改src目录下hibernate.cfg.xml文件的末尾部分，如例7-4所示（同样，需要修改的部分以粗体字显示）。

例7-4: 配置Hibernate使用标注

```
.....
<!--Don't echo all executed SQL to stdout-->
<property name="show_sql">false</property>
<!--disable batching so HSQLDB will propagate errors
correctly.-->
<property name="jdbc.batch_size">0</property>
<!--List all the annotated classes we're using-->
<mapping class="com.oreilly.hh.data.Album"/>
<mapping class="com.oreilly.hh.data.AlbumTrack"/>
<mapping class="com.oreilly.hh.data.Artist"/>
<mapping class="com.oreilly.hh.data.Track"/>
</session-factory>
</hibernate-configuration>
```

为什么必须这么做

你可能会问，为什么需要在配置文件中列出标注过的类？

Hibernate不能通过类的标注而自己找到它们？嗯，可以找到它们，如果你修改编码风格，完全依靠**JPA**接口（使用**JPA EntityManager**的**Hibernate**实现，而不是使用**Hibernate Session**），不用告诉**Hibernate**到哪找标注过的类，**Hibernate**自己就可以找到它们，真令人高兴。但是，如前面所述，本书不打算在标注上做过多介绍。如果坚持使用**Hibernate**的原生接口，就应该显式声明用于执行持久化的类，即使是在使用标注来控制持久化时，也应该这么做。

我们差不多已经做好准备工作了，就差用标注过的类来组成这种方法的核心！接下来就介绍本章真正有趣的部分，看看数据对象的

Java源代码标注长得什么样儿，它是怎么控制Hibernate映射的。

[1] <http://xdoclet.sourceforge.net/xdoclet/index.html>.

为模型对象添加标注

可以在许多Java元素上应用标注。在使用Hibernate时，通常关注的是对类和它的字段进行标注，以指定如何将模型对象映射到数据库模式。这类似于XML映射文档是被映射的类和属性的结构化描述方式。已经进行了足够的背景介绍和解释，接下来就深入主题，以具体的实例（在第6章中开发的完整例子）来看看如何使用标注对类进行映射。

应该怎么做

例7-5演示了一种对Artist类进行标注的方法。本章只是介绍Hibernate Annotations的基础，如果你想更加彻底地了解这些标注，请参考Hibernate Annotations项目网站，它的网址是<http://annotations.hibernate.org>（[\[1\]](#)）。为了节省篇幅，对下载的完整源代码中的标注类进行了一定的压缩，压缩了空白字符，省略了JavaDoc注释。由于这些源代码是手工编写的类，而不是像前几章中用代码生成工具生成的类，所以就有空间为这些代码加入更多、更详细的JavaDoc注释。建议你也看看下载到的源代码，很有价值。

例7-5：标注Artist类

```

package com.oreilly.hh.data;
import java.util.*;
import javax.persistence.*; ❶
import org.hibernate.annotations.Index;
@Entity ❷
@Table (name="ARTIST")
@NamedQueries ({
    @NamedQuery (name="com.oreilly.hh.artistByName",
        query="from Artist as artist where upper (artist.name) =upper (:
name) ")
})
public class Artist{
    @Id ❸
    @Column (name="ARTIST_ID")
    @GeneratedValue (strategy=GenerationType.AUTO)
    private Integer id;
    @Column (name="NAME", nullable=false, unique=true) ❹
    @Index (name="ARTIST_NAME", columnNames={"NAME"}) ❺
    private String name;
    @ManyToMany ❻
    @JoinTable (name="TRACK_ARTISTS",
        joinColumns={@JoinColumn (name="TRACK_ID") },
        inverseJoinColumns={@JoinColumn (name="ARTIST_ID") })
    private Set<Track> tracks;
    @ManyToOne ❼
    @JoinColumn (name="actualArtist")
    private Artist actualArtist;
    public Artist () {}
    public Artist (String name, Set<Track> tracks, Artist
actualArtist) {
        this.name=name;
        this.tracks=tracks;
        this.actualArtist=actualArtist;
    }
    public Integer getId () {return id; }
    public void setId (Integer id) {
        this.id=id;
    }
    public String getName () {return name; }
    public void setName (String name) {
        this.name=name;
    }
    public Artist getActualArtist () {return actualArtist; }
    public void setActualArtist (Artist actualArtist) {
        this.actualArtist=actualArtist;
    }
    public Set<Track>getTracks () {return tracks; }
    public void setTracks (Set<Track>tracks) {

```

```

    this.tracks=tracks;
}
/**
 *Produce a human-readable representation of the artist.
 *
 *@return a textual description of the artist.
 */
public String toString () {
    StringBuilder builder=new StringBuilder ();
    builder.append (getClass () .getName () ) .append ("@" );
    builder.append (Integer.toHexString (hashCode () ) ) .append ("
[");
    builder.append ("name") .append ("='") .append (getName
() ) .append ("");
    builder.append ("actualArtist") .append ("='") .append
(getActualArtist () );
    builder.append ("") .append ("]");
    return builder.toString ();
}
}

```

❶ 为了使用非核心Java语言自身的标注，例如我们在这里讨论的与持久化相关的标注，得先导入它们。标注其实也只是Java类（尽管这些类会实现一个特殊的接口，但它们的声明方法有些意思，相关介绍已经超出本章讨论的范围，有兴趣的话可以阅读第14章），所以使用普通的import语句像这样导入它们就可以了。

如前所述，我们在这里需要使用的大多数标注都是由标准的EJB 3标注（在javax.persistence包中定义）演化而来的。我们之所以需要一个Hibernate特定的标注，是为了能够获得一个特定的索引Maven会自动下载并让我们能够使用Java Persistence API，因为我们在更新过的build.xml中要求它是Hibernate Annotations的一个依赖。我们可以像这样单独地使用Java Persistence API，因为JDK中专门设计让它在Java SE

环境中可以单独使用，在Java EE中也内建了一部分对EJB的支持。如果你想了解更多内容，它的主页（[\[2\]](#)）就是一个好的起点。

❷应用到Artist类上的这组标注是一个整体。Entity标注将这个类标记为可持久化的。Table标注是可选的，标注处理器将为标注提供非常合理的默认假设，但是我们在这里想演示如何明确地指定表名，以防止把错误的名称连接到现有的数据库。

注意：这种查询语言关系表明在很大程度上Hibernate确实影响了EJB 3的发展方向。

那么我们的查询在Java源代码中到底做了什么？唉，这是在使用标注时，Hibernate原生接口表现出来的另一个缺点：没有放置命名查询的地方。如果我们使用JPA EntityManager，就可以将命名查询放在一个persistence.xml文件中，这样就保留了将SQL语句和Java源代码互相分离的优点。由于在本书中我们坚持使用会话接口，这样当我们使用标注而不是XML映射文件时，就丧失了使用命名查询的优点。但是我们仍然可以用HQL编写查询，使用它的所有功能。如果换用JPA接口，就得使用JPAQL（它是HQL的一个子集）。

❸这里演示了如何标注映射属性。这是一个特殊的例子，因为要标注的属性也是对象的惟一标识符，如@Id标注所示。可以用标注指定不同的ID生成策略，就像用Hibernate的XML映射文档一样（标注旨

在完全取代现有O/R层的功能，标准的JPA选择以及Hibernate自己的标注，实际上你差不多可以做想要的任何事情了）。为生成风格选择AUTO，相当于在XML中指定<generator class="native"/>的基于标注的等价物。它告诉Hibernate使用对于数据库来说最自然的任何处理方法。

与实体级的标注一样，你可以省略一些选择（例如列名），默认的选项就相当合理，但我们在这里想演示当你有特殊需要时如何进行配置。事实上，根本不需要为属性添加标注—JPA将假设实体的所有属性都要映射，除非特别指定（要是通过标注的话，自然是：
@Transient用于这一目的）。

也要注意，我们的是将标注加到了实际的字段，而不是访问器方法上。这是告诉Hibernate直接访问字段，而访问器在一个类中适合在运行时为其他类提供抽象，但这样与持久化又不能保持兼容。在许多情况下，你可能想让Hibernate使用访问器方法，只要将标注放到相应的getter或setter方法上就可以了。你需要挑出一个方法或其他方法，但属性之间的混合和匹配还不支持（通过JPA，再加上Hibernate的一些扩展.....即便你能够这样做，也可能引起其他的混乱）。

④当对列进行映射时，可以用许多可选的属性来控制映射结果，例如是否为null、惟一约束等等，就像在XML映射文件中的配置一样。

❸为了能够指定一个列应该有一个索引（以及如何建立索引），是我们在这个例子中增加Hibernate标注的一个原因。`@Index`标签不是标准JPA标注的一部分，它是一个有用的Hibernate扩展。使用这个标注就让我们代码需要依靠Hibernate，但除了这是一本有关Hibernate的书这一事实以外，如前所述（后面也会介绍），还有许多原因可以解释为什么你将经常要做出同样的选择。

❹和映射文档一样，在关联配置上也有更多的选项。在这个例子中，我们描述了一个Track之间的多对多关系，显式地描述出在数据库中如何表达这一关系。

❺有时你不需要在标注中配置很多东西，即使你很清楚这些配置。这个例子是为了演示标注为何要比XML映射文件更加简洁。我们使用`@JoinColumn`标注来设置列名（与基于XML的配置方法一样）。如果没有这个标注，也能正常运行，但是默认的列名稍微有点冗长：`ACTUALARTIST_ARTIST_ID`。

除了以上这段代码，这个类没什么可介绍的了，它是一个简单的数据bean，与前几章中根据映射文档生成的数据类非常相似。下载到的代码中的JavaDoc注释，详细的解释了字段和方法的作用，比前面用工具生成的代码中的注释详细多了。

这个标注过的类生成的ARTIST数据表，与前面几章中根据Artist.hbm.xml映射文档而得到的ARTIST表完全相同。

标注Track类

标注过的Artist类需要引用Track类。例7-6演示了Track类中的标注，其中也引入了一些新的问题。

例7-6：标注Track类

```
package com.oreilly.hh.data;
import java.sql.Time;
import java.util.*;
import javax.persistence.*;
import org.hibernate.annotations.CollectionOfElements;
import org.hibernate.annotations.Index;
@Entity
@Table (name="TRACK")
@NamedQueries ({
    @NamedQuery (name="com.oreilly.hh.tracksNoLongerThan",
        query="from Track as track where track.playTime<=: length")
})
public class Track{
    @Id
    @Column (name="TRACK_ID")
    @GeneratedValue (strategy=GenerationType.AUTO)
    private Integer id;
    @Column (name="TITLE", nullable=false)
    @Index (name="TRACK_TITLE", columnNames={"TITLE"})
    private String title;
    @Column (nullable=false)
    private String filePath;
    @Temporal (TemporalType.TIME) ❶
    private Date playTime;
    @ManyToMany
    @JoinTable (name="TRACK_ARTISTS",
        joinColumns={@JoinColumn (name="ARTIST_ID") },
        inverseJoinColumns={@JoinColumn (name="TRACK_ID") })
```

```

private Set<Artist>artists;
@Temporal (TemporalType.DATE) ❷
private Date added;
@CollectionOfElements❸
@JoinTable (name="TRACK_COMMENTS",
joinColumns=@JoinColumn (name="TRACK_ID") )
@Column (name="COMMENT")
private Set<String>comments;
@Enumerated (EnumType.STRING) ❹
private SourceMedia sourceMedia;
@Embedded❺
@AttributeOverrides ({❻
    @AttributeOverride (name="left", column=@Column
(name="VOL_LEFT") ) ,
    @AttributeOverride (name="right", column=@Column
(name="VOL_RIGHT") )
})
StereoVolume volume;
public Track () {}
public Track (String title, String filePath) {
this.title=title;
this.filePath=filePath;
}
public Track (String title, String filePath, Time playTime,
Set<Artist>artists, Date added, StereoVolume volume,
SourceMedia sourceMedia, Set<String>comments) {
this.title=title;
this.filePath=filePath;
this.playTime=playTime;
this.artists=artists;
this.added=added;
this.volume=volume;
this.sourceMedia=sourceMedia;
this.comments=comments;
}
public Date getAdded () {return added; }
public void setAdded (Date added) {
this.added=added;
}
public String getFilePath () {return filePath; }
public void setFilePath (String filePath) {
this.filePath=filePath;
}
public Integer getId () {return id; }
public void setId (Integer id) {
this.id=id;
}
public Date getPlayTime () {return playTime; }

```

```

public void setPlayTime (Date playTime) {
    this.playTime=playTime;
}
public String getTitle () {return title; }
public void setTitle (String title) {
    this.title=title;
}
public Set<Artist>getArtists () {return artists; }
public void setArtists (Set<Artist>artists) {
    this.artists=artists;
}
public Set<String>getComments () {return comments; }
public void setComments (Set<String>comments) {
    this.comments=comments;
}
public SourceMedia getSourceMedia () {return sourceMedia; }
public void setSourceMedia (SourceMedia sourceMedia) {
    this.sourceMedia=sourceMedia;
}
public StereoVolume getVolume () {return volume; }
public void setVolume (StereoVolume volume) {
    this.volume=volume;
}
public String toString () {❶
    StringBuilder builder=new StringBuilder () ;
    builder.append (getClass () .getName () ) .append ("@" ) ;
    builder.append (Integer.toHexString (hashCode () ) ) .append ("
[") ;
    builder.append ("title" ) .append ("=" ) .append (getTitle
() ) .append ("") ;
    builder.append ("volume" ) .append ("=" ) .append (getVolume
() ) .append ("") ;
    builder.append ("sourceMedia" ) .append ("=" ) .append
(getSourceMedia () ) ;
    builder.append ("") .append ("]") ;
    return builder.toString () ;
}
}

```

❶与日期类型Date（或者是时间戳timestamp，包括时间和日期）相比，由于Java缺少明确的类来表示一天中的时间，我们就需要一种方法来声明如何使用Date类。@Temporal标注提供了这样的功能。在

这个例子中我们标注`playTime`对应于SQL中的TIME列。如果忽略这个标注，默认的列类型将是TIMESTAMP。

❷`added`属性，虽然它与`playTime`具有同样的Date类型，但对应于一个DATE列。

❸`@CollectionOfElements`标注也是一个Hibernate扩展，对于到简单值类型集合的关联，它为我们控制这些类型映射到的表和列提供了一种更简单的方法。这是我们之所以要使用非标准标注的主要原因之一。使用纯JPA，就根本不能直接映射简单值类型（例如String或Integer）的集合。这需要声明一个完整的实体类来持有这样的值，接着再映射这个类。对于那些习惯用Hibernate映射POJO（Plain Old Java Object）的灵活性的人，这可能是一大退步。

❹另一方面，对于内建枚举类型的映射，JPA提供了强大的支持，这是Java 5 enum问世以来给我们带来的一个好处（类型安全的枚举类型模式的广泛采纳成就了这一语言功能）。这个标注比我们在第6章中用过的要更加简单！

❺这是用JPA标注来映射复合自定义类型的标准用法，相当于例6-10。JPA规范要求当我们进行这样的映射时，同时要将StereoVolume类标记为可嵌入的：

```
package com.oreilly.hh.data;
```

```
import java.io.Serializable;
import javax.persistence.Embeddable;
/**
 *A simple structure encapsulating a stereo volume level.
 */
@Embeddable
public class Stereovolume implements Serializable{
.....
}
```

（事实上，**Hibernate**可以很容易地映射嵌套类，而不由我们做这些额外的工作。不过以后的发行版本可能会更加严格地坚持遵守规范，所以按规矩办事并无大碍。）

❹再一次，我们添加了超过实际需要的标注，为的是生成与基于XML的映射版本的例子中完全一样的数据库模式。如果不用`@AttributeOverrides`标注，用于保存两个音量的列将是LEFT和RIGHT（`StereoVolume`类中的属性名称），而不是VOL_LEFT和VOL_RIGHT。

❺为了让我们的测试程序可以打印输出与原来的旧方法一样的信息，我们复制了**Hibernate**代码生成器为我们创建的`toString()`实现。注意，我们可以借这个机会来更新它们，使用`StringBuilder`，而没有必要非得使用线程安全（[\[3\]](#)）（但速度更慢）的`StringBuffer`。

这套标注可以创建TRACK、TRACK_ARTISTS以及TRACK_COMMENTS数据表，与例2-1到例6-10中使用Track.hbm.xml生成的数据库模式一样。

标注Album类

Album类是目前我们构建的其他示例的核心模型类。例7-7演示了如何对它进行标注，以重新创建我们正在处理的数据库模式和映射，同时也引入几个不用再多介绍的概念。

例7-7：标注Album类

```
package com.oreilly.hh.data;
import java.util.*;
import javax.persistence.*;
import org.hibernate.annotations.CollectionOfElements; ❶
import org.hibernate.annotations.Index;
import org.hibernate.annotations.IndexColumn;
@Entity
@Table (name="ALBUM")
public class Album{
    @Id
    @Column (name="ALBUM_ID")
    @GeneratedValue (strategy=GenerationType.AUTO)
    private Integer id;
    @Column (name="TITLE", nullable=false)
    @Index (name="ALBUM_TITLE", columnNames={"TITLE"})
    private String title;
    @Column (nullable=false)
    private Integer numDiscs;
    @ManyToMany (cascade=CascadeType.ALL)
    @JoinTable (name="ALBUM_ARTISTS",
        joinColumns=@JoinColumn (name="ARTIST_ID") ,
        inverseJoinColumns=@JoinColumn (name="ALBUM_ID") )
    private Set<Artist> artists;
    @CollectionOfElements
    @JoinTable (name="ALBUM_COMMENTS",
        joinColumns=@JoinColumn (name="ALBUM_ID") )
    @Column (name="COMMENT")
    private Set<String> comments;
    @Temporal (TemporalType.DATE)
    private Date added;
    @CollectionOfElements❷
```

```

@IndexColumn (name="LIST_POS") ❸
@JoinTable (name="ALBUM_TRACKS",
joinColumns=@JoinColumn (name="ALBUM_ID") )
private List<AlbumTrack>tracks;
public Album () {}
public Album (String title, int numDiscs, Set<Artist>artists,
Set<String>comments, List<AlbumTrack>tracks,
Date added) {
this.title=title;
this.numDiscs=numDiscs;
this.artists=artists;
this.comments=comments;
this.tracks=tracks;
this.added=added;
}
public Date getAdded () {return added; }
public void setAdded (Date added) {
this.added=added;
}
public Integer getId () {return id; }
public void setId (Integer id) {
this.id=id;
}
public Integer getNumDiscs () {return numDiscs; }
public void setNumDiscs (Integer numDiscs) {
this.numDiscs=numDiscs;
}
public String getTitle () {return title; }
public void setTitle (String title) {
this.title=title;
}
public List<AlbumTrack>getTracks () {return tracks; }
public void setTracks (List<AlbumTrack>tracks) {
this.tracks=tracks;
}
public Set<Artist>getArtists () {return artists; }
public void setArtists (Set<Artist>artists) {
this.artists=artists;
}
public Set<String>getComments () {return comments; }
public void setComments (Set<String>comments) {
this.comments=comments;
}
public String toString () {
StringBuilder builder=new StringBuilder () ;
builder.append (getClass () .getName () ) .append ("@" ) ;
builder.append (Integer.toHexString (hashCode () ) ) .append ("
[" ) ;

```



```
builder.append ("title").append ("='").append (getTitle  
( )) .append ("" ) ;  
builder.append ("tracks").append ("='").append (getTracks  
( )) .append ("" ) ;  
builder.append ("]") ;  
return builder.toString ( ) ;  
}  
}
```

❶是的，没有Hibernate特定的扩展，还是不行。在这个类中至少需要3个引用的类。

❷AlbumTrack不是一个实体（它没有ID属性，脱离了Album记录，就不能独立地查询它们的实例）。所以我们使用@CollectionOfElements标注（就像我们前面映射基本类型一样），而不是@OneToMany（用于映射实体）。

❸对于集合映射，JPA和EJB只支持set之类的语义。如第5章所述，能够以特定的顺序来保存记录也很重要，这就是为什么我们要使用像List和array（数组）之类的数据结构，Hibernate可以容易地映射这种有序集合。不过，如果只用JPA，就无法映射这样的集合了！Hibernate的@IndexColumn扩展提供了一种权宜之计。

把这个标注和@JoinColumn信息组合在一起，就可以让Hibernate生成与基于例5-4中的Album.hbm.xml而得到数据库模式完全一样的结果，其中ALBUM_TRACKS表具有一个复合主键（由ALBUM_ID和LIST_POS组成）。

Album类与AlbumTrack类密切相关，AlbumTrack的标注过程如例7-8所示，同样也会重新创建我们的示例数据库模式。

例7-8：标注AlbumTrack类

```
package com.oreilly.hh.data;
import java.io.Serializable;
import javax.persistence.*;
@Embeddable❶
public class AlbumTrack{
    @ManyToOne (cascade=CascadeType.ALL) ❷
    @JoinColumn (name="TRACK_ID", nullable=false)
    private Track track;
    private Integer disc; ❸
    private Integer positionOnDisc;
    public AlbumTrack () {}
    public AlbumTrack (Track track, Integer disc, Integer
positionOnDisc) {
        this.track=track;
        this.disc=disc;
        this.positionOnDisc=positionOnDisc;
    }
    public Track getTrack () {return track; }
    public void setTrack (Track track) {
        this.track=track;
    }
    public Integer getDisc () {return disc; }
    public void setDisc (Integer disc) {
        this.disc=disc;
    }
    public Integer getPositionOnDisc () {return positionOnDisc; }
    public void setPositionOnDisc (Integer positionOnDisc) {
        this.positionOnDisc=positionOnDisc;
    }
    public String toString () {
        StringBuilder builder=new StringBuilder () ;
        builder.append (getClass ().getName () ).append ("@" );
        builder.append (Integer.toHexString (hashCode () ) ).append ("
[") ;
        builder.append ("track" ).append ("=") .append (getTrack
()) .append ("") ;
        builder.append ("]") ;
        return builder.toString () ;
    }
}
```

```
}  
}
```

❶正如前面对Album类映射的讨论，这个类是一个不可以独立存在的实体，所以我们用@Embeddable进行标注，和StereoVolume的映射一样。

❷即使不是实体，我们也需要告诉Hibernate如何处理track属性，这个属性会引用一个实体。如果没有这个标注，试图构建数据库模式就会失败。这个标注也让我们能够保留基于XML的方法中使用同样的列名称。美中不足的是，到Track类的级联请求还不足以在创建新专辑时自动地保存曲目，所以稍后我们将需要回到AlbumTest类的早期版本。在本章的7.3节中，我们将探究一种能够重新获得这种自动级联的模式生成方法。

❸最后这两个属性表明，在使用标注时，有时你根本不需要提供任何标注。虽然这两个属性声明附近什么也没有，但确实能够被映射，标注处理器提供的默认处理可以实现我们想要的映射。

这个类的其他部分相当直接，比其他几个例子看起来更简短，因为它没有需要管理的ID属性，只有其他一些简单属性。

如前所述，这段代码中的映射要求我们在创建新的专辑时，重新负责保存曲目对象。例5-13中Album.hbm.xml最终的映射配置不需要我

们负责保存曲目，所以我们注释掉了AlbumTest.java的addAlbumTrack（）中调用session.save（track）的那行代码。现在我们需要取消对这行的注释，这样才会与例5-8保持一致。

可以有效吗

编辑好所有代码后，接下来就可以创建数据库模式了。例7-9演示了我们现在用这种基于标注的方法，在运行ant schema命令后，得到的大部分结果。其中，为了方便阅读，对创建表格的语句行进行了重新格式化。

例7-9：使用标注过的类来创建数据库模式（重点部分）

```
%ant schema
Buildfile: build.xml
Downloading: org/hibernate/hibernate-
annotations/3.3.0.ga/hibernate-annotations-
3.3.0.ga.pom❶
Transferring 1K
Downloading: org/hibernate/hibernate/3.2.1.ga/hibernate-
3.2.1.ga.pom
Transferring 3K
Downloading: javax/persistence/persistence-api/1.0/persistence-
api-1.0.pom
Transferring 1K
Downloading: org/hibernate/hibernate-commons-
annotations/3.3.0.ga/hibernate-comm
ons-annotations-3.3.0.ga.pom
Transferring 1K
Downloading: org/hibernate/hibernate-
annotations/3.3.0.ga/hibernate-annotations-
3.3.0.ga.jar
Transferring 258K
```

```

    Downloading: javax/persistence/persistence-api/1.0/persistence-
api-1.0.jar
    Transferring 50K
    Downloading: org/hibernate/hibernate-commons-
annotations/3.3.0.ga/hibernate-comm
ons-annotations-3.3.0.ga.jar
    Transferring 64K
    prepare:
    [copy]Copying 1 file to/Users/jim/svn/oreilly/hibernate/current/
examples/ch07/classes
    compile:
    [javac]Compiling 10 source files
to/Users/jim/svn/oreilly/hibernate/
current/examples/ch07/classes
    schema:
    [hibernatetool]Executing Hibernate Tool with a Hibernate
Annotation/EJB3 Config
uration②
    [hibernatetool]1.task: hbm2ddl (Generates database schema)
    [hibernatetool]alter table ALBUM_ARTISTS drop constraint
FK7BA403FCB99A6003;
.....
    [hibernatetool]alter table TRACK_COMMENTS drop constraint
FK105B2688E424525B;
    [hibernatetool]drop table ALBUM if exists;
.....
    [hibernatetool]drop table TRACK_COMMENTS if exists;
    [hibernatetool]create table ALBUM (ALBUM_ID integer generated by
default
as identity (start with 1) , added date, numDiscs integer,
TITLE varchar (255) not null,
primary key (ALBUM_ID) ) ; ③
    [hibernatetool]create table ALBUM_ARTISTS (ARTIST_ID integer not
null,
ALBUM_ID integer not null,
primary key (ARTIST_ID, ALBUM_ID) ) ;
    [hibernatetool]create table ALBUM_COMMENTS (ALBUM_ID integer not
null,
COMMENT varchar (255) ) ;
    [hibernatetool]create table ALBUM_TRACKS (ALBUM_ID integer not
null,
disc integer, positionOnDisc integer, TRACK_ID integer,
LIST_POS integer not null,
primary key (ALBUM_ID, LIST_POS) ) ;
    [hibernatetool]create table ARTIST (ARTIST_ID integer generated
by default
as identity (start with 1) , NAME varchar (255) not null,
actualArtist integer,

```

```

        primary key (ARTIST_ID) , unique (NAME) ) ;
[hibernatetool]create table TRACK (TRACK_ID integer generated by
default
as identity (start with 1) , added date,
filePath varchar (255) not null, playTime time,
sourceMedia varchar (255) , TITLE varchar (255) not null,
VOL_LEFT smallint, VOL_RIGHT smallint,
primary key (TRACK_ID) ) ;
[hibernatetool]create table TRACK_ARTISTS (ARTIST_ID integer not
null,
TRACK_ID integer not null,
primary key (TRACK_ID, ARTIST_ID) ) ;
[hibernatetool]create table TRACK_COMMENTS (TRACK_ID integer not
null,
COMMENT varchar (255) ) ;
[hibernatetool]create index ALBUM_TITLE on ALBUM (TITLE) ;
[hibernatetool]alter table ALBUM_ARTISTS add constraint
FK7BA403FCB99A6003 fore
ign key (ARTIST_ID) references ALBUM;
.....
[hibernatetool]alter table TRACK_COMMENTS add constraint
FK105B2688E424525B for
eign key (TRACK_ID) references TRACK;
[hibernatetool]9 errors occurred while performing<hbm2ddl>.❹
[hibernatetool]Error#1: java.sql.SQLException: Table not found:
ALBUM_ARTISTS
in statement[alter table ALBUM_ARTISTS]
.....
BUILD SUCCESSFUL
Total time: 5 seconds

```

❶因为这是我们第一次要求Maven Ant Tools提供Hibernate Annotations，它们将作为依赖而自动下载。

❷这里你可以看到，正在使用标注来驱动数据库模式的创建。

❸如果同例5-7中的相应行进行比较，你会发现虽然显示的顺序不同，但列定义是完全一样的。ALBUM_ARTISTS、ALBUM_COMMENTS以及最具挑战性的ALBUM_TRACKS的定义都

是同样的情况。我们已经成功地用标注再次创建了原来的数据库模式。

④在创建数据库模式，当运行时根本不存在数据库时，就会看到这些普通的错误信息。这些错误不会中止创建过程，虽然有错误，但可以认为创建过程还是成功的。

同时，你可以在本章目录下运行`ant db`命令，就像第5章做的那样，并比较一下它们各自生成的数据库模式。当然，看看实际的数据，效果应该更好。为了让`CreateTest.java`可以处理基于标注的映射，需要修改它的两个地方，如例7-10所示。我们需要导入`AnnotationConfiguration`类，并在以前使用`Configuration`类的位置使用它。

例7-10：对测试类进行调整，以处理基于标注的映射

```
package com.oreilly.hh;
import org.hibernate.*;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;
.....
public static void main (String args[]) throws Exception{
//Create a configuration based on the annotations in our
//model classes.
Configuration config=new AnnotationConfiguration ();
config.configure ();
.....
}
```

修改完后，运行`ant ctest`命令就可以创建示例数据，并使用多个`ant db`的实例，依次对数据进行比较。

对QueryTest.java、QueryTest2.java以及AlbumTest.java进行同样的修改。因为运行ant qtest和ant qtest2命令的输出与前一章相同，这里就不再演示了。但是我们想演示来自AlbumTest的输出内容，因为它需要依赖整个数据库模式，所以可以作为一个不错的完整性检查。用它也可以验证例7-8后面介绍的那行用于保存曲目的代码确实被取消注释了。运行ant atest，对我们这个基于标注的数据库模式进行测试的结果如例7-11所示。

例7-11：运行AlbumTest，测试标注的使用

```
atest:
[java]com.oreilly.hh.data.Album@27d19d[title='Counterfeit
e.p.'tracks=[
  com.oreilly.hh.data.AlbumTrack@bf4c80[track='com.oreilly.hh.data
.Track@2e3919[
  title='Compulsion'volume='Volume[left=100,
right=100]'sourceMedia='CD']], c
  om.oreilly.hh.data.AlbumTrack@3778cf[track='com.oreilly.hh.data.
Track@f4d063[t
  itle='In a Manner of Speaking'volume='Volume[left=100,
right=100]'sourceMedia=
  'CD']],
com.oreilly.hh.data.AlbumTrack@dc696e[track='com.oreilly.hh.data.Tr
a
ck@a5dac0[title='Smile in the Crowd'volume='Volume[left=100,
right=100]'sourc
eMedia='CD']],
com.oreilly.hh.data.AlbumTrack@8dbef1[track='com.oreilly.hh.d
ata.Track@c4b579[title='Gone'volume='Volume[left=100,
right=100]'sourceMedia=
  'CD']],
com.oreilly.hh.data.AlbumTrack@f2f761[track='com.oreilly.hh.data.Tr
a
ck@8cd64[title='Never Turn Your Back on Mother
Earth'volume='Volume[left=100,
right=100]'sourceMedia='CD']],
com.oreilly.hh.data.AlbumTrack@4f1541[track=
```



```
'com.oreilly.hh.data.Track@c042ba[title='Motherless  
Child'volume='Volume[left=  
100, right=100]'sourceMedia='CD']']']']
```

除了Java随机将类加载到的内存地址不同以外，例7-11与例5-10中看到的输出完全相同，和我们希望的一样。

注意：这种方法能行！真的能行！

[1] 如果你跳过本书前面的内容直接阅读这一章，那么最好回去再了解一下用到的这些类和它们之间的关系，至少要浏览一下从第3章开始的内容，再继续学习。这样你的收获可能更多。

[2] <http://java.sun.com/javaee/technologies/persistence.jsp>.

[3] 变量**builder**是方法的局部变量，所以不可能会有多个线程同时使用这一变量。

另一种方法

这个试验表明标注确实是一种映射模型类的可行方法。通过为数不多的几步，我们就可以精确地维护前面章节中逐步建立的数据库模式，只是标注这种方法不能让我们在创建Album时级联创建Track对象。如果以稍微不同的方式来考虑AlbumTrack类，则还有另一种方法，让我们在数据库模式中能够维护这种自动的级联处理，同时也提供了一些其他功能。

将AlbumTrack映射为一个完整的实体，我们就可以添加级联标注，Hibernate就会优先处理Album定义，以嵌入对Track的引用。这也给我们带来一些新的需要考虑的复杂问题，但其中一部分需要视不同的时机而定夺。首先，将AlbumTrack作为实体就需要具有ID。同时，因为我们接着需要不从Album开始就可以处理AlbumTrack对象，所以应该扩充AlbumTrack模型，以提供从ALBUM_TRACKS表返回到ALBUM表的链接（我们将它用于Hibernate复合键）。为此，需要增加一个album属性。例7-12演示了AlbumTrack映射中关键的一部分，也是这种方法的独特之处所在。

例7-12：将AlbumTrack类标注为实体

```
package com.oreilly.hh.data;
import java.io.Serializable;
```

```

import javax.persistence.*;
@Entity❶
@Table (name="ALBUM_TRACKS")
public class AlbumTrack{
    @Id❷
    @GeneratedValue (strategy=GenerationType.AUTO)
    private Integer id;
    @ManyToOne
    @JoinColumn (name="ALBUM_ID", insertable=false, updatable=false,
❸
    nullable=false)
    private Album album;
    @ManyToOne (cascade=CascadeType.ALL) ❹
    @JoinColumn (name="TRACK_ID", nullable=false)
    private Track track;
    .....
    public Integer getId () {
        return id;
    }
    public void setId (Integer id) {
        this.id=id;
    }
    public Album getAlbum () {
        return album;
    }
    ❺
    public Track getTrack () {
        .....
    }
}

```

❶明显地，我们将类标注由@Embeddable修改为@Entity，而且就在这里选择了表名，而不是在Album类的源文件中。

❷此处与基于XML的例子中的模式差别最大。以前我们的ALBUM_TRACK表使用一个复合主键，利用的是ALBUM_ID和TRACK_ID的一个特定组合在数据库中只有一个记录这一事实，这样就节省了我们在ALBUM_TRACK表中提供一个单独的ID列的需要。

虽然让AlbumTrack成为一个实体后还保留原来的数据库模式也是可能的，不过这需要付出不少的努力。JPA要求所有的复合主键都要被映射为一个单独的类，由这个类来提供一个主键的各个成员。所以，为了保留原来我们的数据库模式，就必须对模型类进行比较大的修改，只是为了持有AlbumTrack类的主键而创建一个新的类。

相反，也可以稍微修改一下数据库模式，为ALBUM_TRACKS增加一个ID列（应该承认，这个列没有什么实际用途），看起来是一种破坏力比较小的选择。无论如何，当改变映射方法时，我们将要面对各种权衡，这个例子就是一个有趣的演示。

❸到Album的映射是我们以前见过的@ManyToOne，但是还需要提供一些额外的参数，才能让它按我们想要的方式来工作。这段“咒语”用于重新创建我们曾经用Album.hbm.xml取得的映射效果，让Hibernate完全控制对ALBUM_TRACKS表的ALBUM_ID列的维护。如果没有@JoinColumn标注中的insertable和updatable属性，我们就得必须修改AlbumTest，为每个AlbumTrack对象显式地设置album属性，这样也就意味着失去了一些我们想要从Hibernate得到的自动处理。

当使用实体上的索引映射时（具有@IndexColumn或@MapKey），你应该记住这种应用模式。

④既然AlbumTrack是一个实体，Hibernate就能够为它的track属性应用cascade的设置。

⑤我们可以强化一个概念，Hibernate在管理到专辑的链接（album属性）时，并没有使用setAlbum（）方法。

当然，在Album.java中这一关系的映射方式也会有稍微的不同，如例7-13所示。

例7-13: Album.java中的AlbumTracks实体映射

```
.....
@OneToMany (cascade=CascadeType.ALL)
@IndexColumn (name="LIST_POS")
@JoinColumn (name="ALBUM_ID", nullable=false)
private List<AlbumTrack> tracks;
.....
```

@OneToMany标注中的级联（cascade）设置，它通过Album中嵌入的Track引用来建立这种完整的级联关系，我们在例5-13开发的Album.hbm.xml最终版本中也曾经建立过这样的关联，在那个例子中是由专辑对象来管理它们的曲目对象的生命周期。在这里，需要再一次将AlbumTest的addAlbumTrack（）方法中用于保存曲目的那行代码注释掉。这样，我们用不同的数据库模式重新创建了原来的功能。如果不怕麻烦，也可以创建一个类来负责管理复合键，这样就可以保留原来的功能和数据库模式。

和许多其他Hibernate API（以及一般的面向对象的建模方法）一样，可以用多种方法来实现一个功能。

现在怎么办

希望这一章可以让你对如何用标注来表达数据映射有个大致的了解，也希望当你为自己的项目探索新的选择时，本章介绍的内容可以作为一个良好的起点。在本书接下来的部分，我们将不列举相关技术的所有细节和功能（这应该是Hibernate参考手册的任务），虽然有时介绍的比较肤浅，但希望我们讨论的一些问题可以作为帮助你解决模糊问题的示例。如果所有办法都解决不了问题，那只能尝试让问题重新出现，再将错误消息粘贴到Google上搜索！或者，如果你是好“公民”的话，可以研究一下源代码，将问题发布到Hibernate论坛上以寻求帮助，把解决问题的希望留给未来的用户，并帮助突出应该加强Hibernate文档的哪些部分。

在接下来的几章中，我们将继续回到基于XML的世界，看看几种查询数据的方法。但是，还应该记住标注的概念，在本书结尾部分介绍Spring和Stripes的章节中，将会再次用到它们。

第8章 条件查询

像HQL（以及作为它的基础的SQL）这样的关系型查询语言都非常灵活而且功能强大，但是如果要真正精通，也得花费很长的时间。很多应用程序开发人员对SQL只有基本的了解，只能根据以往的项目模仿些相似的示例，当碰上真正没有遇到过的或是非常难以理解的查询表达式时，才会寻求数据库专家的帮助。

将查询语言的语法和Java代码混杂在一起，也很麻烦。第3.4节介绍了一种将所有查询语句单独放在另一个文件中，可以集中对它们进行查看和编辑，不需要使用Java字符串转义字符序列（`escape sequence`）和串联（`concatenation`）语法。不过，即使采用这种技巧，也是直到加载映射文档时才会解析HQL查询语句，也就是说，HQL查询内容中隐藏的语法错误在应用程序运行以前都无法捕获。

Hibernate采用条件查询的方法，为这些问题提供了一种不同寻常的解决方案。这种方法通过创建简单的Java对象，并把它们串连起来，将其作为过滤器来筛选出你想要的结果。你可以建立嵌套的、结构化的表达式。这种机制也可以让你只提供示例对象，以表明你想查找的内容是什么，同时还能控制哪些细节需要关注、哪些属性可以忽略。

从后面的介绍中可以看到，这种功能非常方便。但是坦率地讲，它也有自身的（非常次要的）缺点。把冗长的查询表达式转换成Java API会占用更多的内存空间，对于经验丰富的数据库开发人员而言，他们对条件查询不像对类SQL（SQL-like）查询语言那么熟悉。有些东西你无法用以前的条件查询API来加以表达，诸如投影（从一个类的多个属性中取出子集，例如"select title, id from com.oreilly.hh.Track"，而不是"select*from com.oreilly.hh.Track"）和聚合（aggregation）（对查询结果做统计总结，例如获取某个属性的总和、平均值以及总数）。这种非常严重的不足，在编写本书第1版时API就存在，不过，在Hibernate 3中已经得到了解决。我们会向你介绍现在应该怎么实现这些条件查询。下一章还会演示如何使用Hibernate的面向对象的查询语言（HQL）来完成此类任务。

不论使用哪一种Hibernate的方法来表达查询，最终都会生成特定数据库的SQL语句，由SQL来实现查询目的。所幸，你不会看到这些底层细节，但是，如果你对这些细节感兴趣的话，可以使用Hibernate配置文件的show_sql属性打开SQL日志输出（如例3-1所示），或是在Eclipse中使用交互式的SQL查询预览（将在第11章介绍）。

使用简单条件查询

我们先建一个条件查询来查找播放时间少于指定长度的曲目，将例3-11中所用的HQL替换掉，并更新例3-12的代码。

应该怎么做

需要明白的第一件事就是如何指定我们想要检索的对象的类型。在建立条件查询时，不会涉及任何查询语言。相反，你得构建一个由Criteria对象组成的树状结构来描述你需要检索的对象。Hibernate Session就是创建这些Criteria对象的工厂，而你需要做的就是指定想要检索到的对象的类型，够方便的吧。

编辑QueryTest.java，将tracksNoLongerThan () 方法的内容替换成例8-1所示的内容。

注意：这些示例假设已经按照前几章所述建立好了数据库。如果你不想从头开始，就下载示例代码，然后再跳到这一章的目录，运行codegen、schema、以及ctest这几个构建目标。即使你按照书中介绍一路走来，运行schema和ctest也将确保生成这些示例展示的数据。

例8-1：条件查询入门

```
public static List tracksNoLongerThan (Time length, Session
session) {
    Criteria criteria=session.createCriteria (Track.class) ;
    return criteria.list () ;
}
```

会话对象的`createCriteria()`方法会创建一个条件查询对象（`Criteria`类的实例），由它返回作为参数传递给它的持久化类的所有实例，真够简单的。当然，如果现在运行示例，会看到数据库中保存的所有曲目，因为我们还没有使用任何查询条件（`criteria`）来限制查询结果（如例8-2所示）。

例8-2：羽翼未丰的条件查询将返回所有曲目

```
%ant qtest
.....
qtest:
[java]Track: "Russian Trance" (PPK) 00: 03: 30, from Compact Disc
[java]Track: "Video Killed the Radio Star" (The Buggles) 00: 03:
49,
from VHS Videocassette tape
[java]Track: "Gravity's Angel" (Laurie Anderson) 00: 06: 06, from
Compact
Disc
[java]Track: "Adagio for Strings (Ferry Corsten Remix) " (Ferry
Corsten,
William Orbit, Samuel Barber) 00: 06: 35, from Compact Disc
[java]Track: "Adagio for Strings (ATB Remix) " (William Orbit,
ATB,
Samuel Barber) 00: 07: 39, from Compact Disc
[java]Track: "The World'99" (Ferry Corsten, Pulp Victim) 00: 07:
05,
from Digital Audio Stream
[java]Track: "Test Tone 1"00: 00: 10
[java]Comment: Pink noise to test equalization
```

好，够简单。那么，应该怎么挑选出我们想要的曲目？也很简单！只要在源文件顶端新增加一行：

```
import org.hibernate.criterion.*;
```

然后，再在这个方法中新增加一行，如例8-3所示。

例8-3：用Criteria查询完全取代第3章使用的HQL查询

```
public static List tracksNoLongerThan (Time length, Session
session) {
    Criteria criteria=session.createCriteria (Track.class) ;
    criteria.add (Restrictions.le ("playTime", length) ) ;
    return criteria.list () ;
}
```

注意：就像HQL那样，表达式总是以对象属性来表示，而不使用数据表的字段。

Restrictions类是一个工厂类，用于获取在查询中指定各种约束条件的**Criterion**实例。它的`le ()`方法会创建一个**Criterion**对象，限制一个属性必须小于或等于指定的值。在这个例子中，我们想让**Track**的**playTime**属性不大于传递给该方法的值。最后再把它添加到所要的条件查询组合中。下一节我们将会看到其他几个通过**Restrictions**能用的**Criterion**类型。附录B会列出全部的**Criterion**类型，如果你想支持新的条件形式，也可以自行创建**Criterion**接口的实现。

这次运行查询时，会得到长度不超过7分钟的曲目，如例8-4所示。

例8-4：完整的简单条件查询的结果

```
%ant qtest
.....
qtest:
[java]Track: "Russian Trance" (PPK) 00: 03: 30, from Compact Disc
[java]Track: "Video Killed the Radio Star" (The Buggles) 00: 03:
49,
from VHS Videocassette tape
[java]Track: "Gravity's Angel" (Laurie Anderson) 00: 06: 06, from
Compact Disc
[java]Track: "Adagio for Strings (Ferry Corsten Remix) " (Ferry
Corsten,
Samuel Barber, William Orbit) 00: 06: 35, from Compact Disc
[java]Track: "Test Tone 1"00: 00: 10
[java]Comment: Pink noise to test equalization
```

注意：本书新版修订了这个示例，"Adagio for Strings"会在iTunes中随机开始播放。

在实际应用中，绝大多数用于检索对象的查询都是非常简单的，而条件查询更是用Java表达这些查询时极为自然和简洁的方式。新的 `tracksNoLongerThan ()` 方法实际上比例3-12和那个需要将查询添加到映射文件的示例（例3-11）还要短！不过，它们最终都以相同的模式来访问底层数据库，所以运行效率相当。

不过，如果需要的话，也可以把代码写得更紧凑些。`Add ()` 和 `createCriteria ()` 方法会返回Criteria实例，所以可以在同一Java语句中连续操作该实例。利用这一点，我们就能把这个方法再归纳一下，变成例8-5的样子。

例8-5：更为紧凑的条件查询

```
public static List tracksNoLongerThan (Time length, Session
session) {
    return session.createCriteria (Track.class) .
    add (Restrictions.le ("playTime", length) ) .list () ;
}
```

选择编码风格就是在空间和可读性之间做出取舍（不过，有些人会觉得这种紧凑、连续编码的版本更具可读性）。

其他

对结果进行排序？从所有匹配的对象中取回它们的一部分？和Query接口一样，Criteria接口也可以让你调用setMaxResults () 和 setFirstResult () 方法来限制取回的结果数目（以及选择从哪里开始）。此外，这个接口也可以让你控制结果返回的顺序（在HQL查询中，则是使用order by子句），如例8-6所示。

例8-6：按照标题对结果进行排序

```
public static List tracksNoLongerThan (Time length, Session
session) {
    Criteria criteria=session.createCriteria (Track.class) ;
    criteria.add (Restrictions.le ("playTime", length) ) ;
    criteria.addOrder (Order.asc ("title") .ignoreCase () ) ;
    return criteria.list () ;
}
```

Order类只是表达排列顺序的一种方式。它有两个静态的工厂方法：asc () 和desc () 方法，分别用于创建升序和降序的排列。每个

方法都以要排序的属性名称作为参数。如果调用Order实例的ignoreCase () 方法，则排序将不区分字母的大小写，这经常是你想要的显示方式。运行这个版本的示例，其结果如例8-7所示。

例8-7：对检索结果进行排序

```
%ant qtest
.....
qtest:
[java]Track: "Adagio for Strings (Ferry Corsten Remix) " (Ferry
Corsten,
Samuel Barber, William Orbit) 00: 06: 35, from Compact Disc
[java]Track: "Gravity's Angel" (Laurie Anderson) 00: 06: 06, from
Compact
Disc
[java]Track: "Russian Trance" (PPK) 00: 03: 30, from Compact Disc
[java]Track: "Test Tone 1"00: 00: 10
[java]Comment: Pink noise to test equalization
[java]Track: "Video Killed the Radio Star" (The Buggles) 00: 03:
49,
from VHS Videocassette tape
```

可以在Criteria添加多个Order，这样Criteria就会按每个Order的先后进行排序（先按第一个Order排序，如果有任何查询结果与那个属性具有相同的值，就再按第二个Order进行排序，依此类推）。

组合式条件查询

可以想到，如果在查询中添加多个**Criterion**，那么结果中的对象就得满足所有的**Criterion**。这相当于使用**Restrictions.conjunction ()** 建立一个组合条件，详情可以参阅附录B。就例8-8来说，我们可以限制查询结果，所以在该方法中另加一行，使得曲目的标题中必须包含字母"A"。

例8-8：一个较挑剔的条件查询

```
Criteria criteria=session.createCriteria (Track.class) ;
criteria.add (Restrictions.le ("playTime", length) ) ;
```

```
criteria.add (Restrictions.like ("title", "%A%") ) ;
criteria.addOrder (Order.asc ("title") .ignoreCase () ) ;
return criteria.list () ;
```

准备好以后，运行程序，这次会得到较少的结果，如例8-9所示。

例8-9：播放时间等于或少于7分钟且标题包含字母"A"的曲目

```
qtest:
[java]Track: "Adagio for Strings (Ferry Corsten Remix) " (Samuel
Barber,
Ferry Corsten, William Orbit) 00: 06: 35, from Compact Disc
[java]Track: "Gravity's Angel" (Laurie Anderson) 00: 06: 06, from
Compact Disc
```

如果你不记得（或不知道）像这样的SQL“%”字符串匹配的语法，**Hibernate**还提供了一个可供调用的参数变量，用于在**Java**中表达你需要的任何匹配。就这个例子来说，可以写成**Restrictions.like**（"title", "A", **MatchMode.ANYWHERE**）。如果你想进行区分大小写的匹配，应该使用**ilike**，而不是**like**。

如果你希望找出满足任意条件之一的对象，而非满足所有条件的对象，就得显式的使用**Restrictions.disjunction**（）将这些条件分组。可以使用**Restrictions**类提供的**Criteria**工厂来建立这些分组以及其他复杂层次的组合。详情可以参阅附录B。例8-10演示了我们对示例查询的修改，让返回的曲目既满足时间长度限制又满足标题包含大写字母A。

注意：条件查询结合了强大的功能和方便性，令人吃惊。

例8-10：限制较宽松的条件查询

```
Criteria criteria=session.createCriteria (Track.class) ;
Disjunction any=Restrictions.disjunction () ;
any.add (Restrictions.le ("playTime", length) ) ;
any.add (Restrictions.like ("title", "%A%") ) ;
criteria.add (any) ;
criteria.addOrder (Order.asc ("title") .ignoreCase () ) ;
return criteria.list () ;
```

这会让我们多获取一个新的"Adagio for Strings"曲目（如例8-11所示）。

例8-11: 标题含有字母"A"或者时间不超过7分钟的曲目

```
qtest:
[java]Track: "Adagio for Strings (ATB Remix) " (ATB, William
Orbit,
Samuel Barber) 00: 07: 39, from Compact Disc
[java]Track: "Adagio for Strings (Ferry Corsten Remix) " (Ferry
Corsten,
William Orbit, Samuel Barber) 00: 06: 35, from Compact Disc
[java]Track: "Gravity's Angel" (Laurie Anderson) 00: 06: 06, from
Compact
Disc
[java]Track: "Russian Trance" (PPK) 00: 03: 30, from Compact Disc
[java]Track: "Test Tone 1"00: 00: 10
[java]Comment: Pink noise to test equalization
[java]Track: "Video Killed the Radio Star" (The Buggles) 00: 03:
49,
from VHS Videocassette Tape
```

最后要注意的是，把这个方法精简到一个表达式也是可行的（如例8-12所示）。这得多亏这些方法的设计精巧的返回值。

例8-12: 代码精简得有些过头

```
return session.createCriteria (Track.class) .add
(Restrictions.disjunction () .
add (Restrictions.le ("playTime", length) ) .
add (Restrictions.like ("title", "%A%") ) ) .
addOrder (Order.asc ("title") .ignoreCase () ) .list () ;
```

虽然得到的结果都一样，但我觉得你也会认为这样做对该方法代码的可读性没有什么帮助（可能LISP专家不算在内吧）！

可以用Restrictions提供的工具来建立各种各样的多重条件。有些情况还得需要HQL，而且超过一定的复杂度的话，可能还是用HQL比较好。但是，条件查询能够让你做很多事情，这常常是正确的做法。

投影和聚合的条件查询

如果你对SQL比较熟悉，那么应该明白本节标题的意思。如果不明白的话，也不要担心，它们其实相当简单。投影只是说，你并不是需要一个表中的所有信息，而是只需要其中的一部分。在像Hibernate这样的面向对象的环境中，投影就是指不必检索回一个完整的对象，只需要对象的一两个属性。聚合就是标识一些属性，并计算针对这些属性的统计信息，例如计算总和、最大值、最小值以及平均值。

在Hibernate 3以前，不使用HQL就没办法进行这样的操作，所以这是Hibernate 3对Criteria API的一个很好的扩展。我们先来看看一些示例。先来个简单的例子，假设我们想打印所有标题包含字母"v"的曲目的标题，但不必加载任何一个完整的Track对象。

应该怎么做

例8-13演示了一个方法，它使用Criteria API提供的投影功能来实现这一目的。

例8-13：对单一属性的简单投影

/**

```
*Retrieve the titles of any tracks that contain a particular
text string.
*
@param text the text to be matched, ignoring case, anywhere in
the title.
@param session the Hibernate session that can retrieve data.
@return the matching titles, as strings.
*/
public static List titlesContainingText (String text, Session
session) {
    Criteria criteria=session.createCriteria (Track.class) ;
    criteria.add (Restrictions.like ("title", text,
MatchMode.ANYWHERE) .❶
    ignoreCase () ) ;
    criteria.setProjection (Projections.property ("title") ) ; ❷
    return criteria.list () ;
}
```

❶这行演示了使用**MatchMode**接口来避免执行字符串处理，也不用为了指定想要的字符串匹配模式而记住特定的“%”字符的用法。

❷这里使用**Projections**类来告诉**criteria**，我们想要进行一个投影操作，我们对取回找到的曲目的**title**属性特别感兴趣。

像我们的其他条件查询一样，这个方法返回的也是一个**List**对象。但是确切地说，到底是什么内容的列表呢？**Criteria**是在**Track**类的基础上创建的，但是由于我们只需要检索回曲目的标题属性，所以构造并返回整个**Track**对象并没有多大意义。事实上，在这种情况下，只要将整个对象投影到一个属性，就可以得到与该属性相关联的类型的一个列表。在这个例子中，因为它是一个字符串属性，所以得到的就是一个包含**String**实例的**List**对象。

注意：这样做确实有意义！

为了检验效果，可以将这个方法添加到QueryTest.java中，修改main（）函数以调用它，如下所示：

```
System.out.println (titlesContainingText ("v", session) );  
运行这个版本的程序，将产生以下输出：  
qtest:  
[java][Video Killed the Radio Star, Gravity's Angel]
```

很显然，通过投影也可以检索回那些不在Restrictions表达式中的各属性。如果我们想得到曲目长度，可以这样做：

```
criteria.setProjection (Projections.property ("playTime") );
```

它的输出结果是：

```
qtest:  
[java][00: 03: 49, 00: 06: 06]
```

当然，方法名称看起来好像不对，输出结果也不明了。如果我们既想取回标题，也想取回长度，那该怎么办？其实也很简单，如例8-14所示。

例8-14：投影到两个属性

```
/**  
*Retrieve the titles and play times of any tracks that contain a  
*particular text string.
```

```

*
@param text the text to be matched, ignoring case, anywhere in
the title.
@param session the Hibernate session that can retrieve data.
@return the matching titles and times wrapped in object arrays.
*/
public static List titlesContainingTextWithPlayTimes (String
text,
Session session) {
Criteria criteria=session.createCriteria (Track.class) ;
criteria.add (Restrictions.like ("title", text,
MatchMode.ANYWHERE)
.ignoreCase () ) ;
criteria.setProjection (Projections.projectionList () .❶
add (Projections.property ("title") ) .❷
add (Projections.property ("playTime") ) ) ;
return criteria.list () ;
}

```

❶projectionList () 方法创建一个ProjectionList实例，它可以包含应用于一个条件查询的多个投影选择。注意，我们正在使用的是例8-5介绍的简洁的链式标记法，这样就不需要再声明一个变量来持有这个实例了。

❷接着，我们只要将所有需要的投影添加到ProjectionList，再将这个ProjectionList实例传递给条件查询对象的setProjection () 方法。

这里最不容易处理的是从查询返回的结果。和前面的一样，它也是一个List，但现在每个列表元素要包含多个值，而且这些值的类型还可能不同。Hibernate采用的办法是返回一个对象数组的列表。以下这段代码用于显示返回的列表，看起来有些复杂：

```
for (Object o: titlesContainingTextWithPlayTimes ("v", session) )
{
    Object[]array= (Object[]) o;
    System.out.println ("Title: "+array[0]+
        " (Play Time: "+array[1]+' ' ) ;
}
```

它将输出以下内容:

```
qtest:
[java]Title: Video Killed the Radio Star (Play Time: 00: 03: 49)
[java]Title: Gravity's Angel (Play Time: 00: 06: 06)
```

确实，这段代码看起来很丑，你绝对不应该在现实中按这样的方式来使用投影。对象/关系映射系统的要点就是你可以只返回对象，在需要一些属性时，再用这些对象来得到你要的属性。这样的话，为什么投影要支持多个值？嗯，事实上是存在好的理由的，它与我们在本节开始介绍的“聚合”的概念有关。很多时候在使用投影时，经常会得到原来根本不直接属于任何对象的值，只是这些值会基于某些对象的属性。例8-15演示了一个方法，它会输出数据库中的每种曲目来源媒介，以及来自这种媒介的所有曲目的数量，还有这种媒介的曲目的最长播放时间。

例8-15：带有聚合的投影

```
/**
 *Print statistics about various media types.
 *
 *@param session the Hibernate session that can retrieve data.
 */
```

```
public static void printMediaStatistics (Session session) {  
    Criteria criteria=session.createCriteria (Track.class) ;  
    criteria.setProjection (Projections.projectionList () .❶  
    add (Projections.groupProperty ("sourceMedia") ) .❷  
    add (Projections.rowCount () ) .❸  
    add (Projections.max ("playTime") ) ) ; ❹  
    for (Object o: criteria.list () ) {❺  
        Object[]array= (Object[]) o;  
        System.out.println (array[0]+"track count: "+array[1]+  
        "; max play time: "+array[2]) ;  
    }  
}
```

注意：现在我们正在利用数据库的功能做些很有意思的事！

这个例子涉及我们目前为止见到的许多事情：

❶和以前一样，我们正在创建一个**ProjectionList**实例来持有想要查询返回的各个项。

❷**groupProperty ()** 方法与我们目前使用过的**property ()** 方法类似，但它是告诉**Hibernate**将指定的属性对记录进行分组，所有值相同的记录就成为结果集中的一条记录。分组是执行聚合操作的关键，能让我们为投影增加聚合值。

❸**rowCount ()** 投影不需要任何参数，因为它只是返回分组到当前结果集中的记录的总数（基于我们的**groupProperty ()** 的值，**sourceMedia**）。这就是我们计算属于每种来源媒介类型的曲目数量的方法。

④`max()` 投影返回分组的结果集中某个属性的最大值。

⑤最后，我们用类似前面例子中创建的输出循环，循环遍历条件查询返回的Object数组的List，并输出它们。

在QueryTest.java的main()函数中调用这个方法很简单：

```
printMediaStatistics(session);
```

它会生成以下输出：

```
qtest:
[java]CD track count: 4; max play time: 00: 07: 39
[java]VHS track count: 1; max play time: 00: 03: 49
[java]STREAM track count: 1; max play time: 00: 07: 05
[java]null track count: 1; max play time: 00: 00: 10
```

这一结果应该会让你对投影和聚合的真正价值有所了解（`null`媒介类型供我们测试使用）。

不过，能够看到，这一输出并没有按任何顺序进行排序。我们可能想进行排序，但是你怎么对投影结果排序呢？答案就是我们需要看看Criteria API中的另一个改进。你可以为对象和属性分配“别名”，以供我们处理使用（就像在数据库查询语言中为表和列起的别名一样），而不论它们是直接来自数据库，还是来自投影和聚合操作。这样，我们按照来源媒介进行排序就容易了，只需要为分组后的属性添加一个别名：

```
add (Projections.groupProperty ("sourceMedia") .as ("media")) .
```

这条语句声明了一个"media"别名，接着就可以通过这一别名进行排序，就像我们前面对普通的对象属性进行排序一样：

```
criteria.addOrder (Order.asc ("media")) ;
```

经过这些变化后，我们就可以看到排序后的输出内容：

```
qtest:  
[java]null track count: 1; max play time: 00: 00: 10  
[java]CD track count: 4; max play time: 00: 07: 39  
[java]STREAM track count: 1; max play time: 00: 07: 05  
[java]VHS track count: 1; max play time: 00: 03: 49
```

之所以使用别名还有些其他原因，使用投影也还可以完成更多的事情，但是接下来要介绍条件查询的其他方面。在本章最后，我们将介绍到哪里可以学习到更多的相关内容。

在关联中应用条件查询

到目前为止，在条件查询的构建上，我们看到的都是查询同一个类的各种属性。当然，在真实的系统中，对象之间有丰富的关联关系。有时，我们想用于过滤结果的细节是来源于这些关联。幸好，条件查询API提供了一种相当简明的方式可以执行这样的搜索。

应该怎么做

假设我们想找出与特定艺人相关联的所有曲目。我们需要的查询就是检查每个Track对象的artists属性中包含的值。artists属性是一个集合，包含与某个曲目相关的所有Artist对象的关联。为了让查询过程更有趣些，再假设我们想找到姓名属性匹配特定子字符串（substring）的艺人相关联的所有曲目。

我们在QueryTest.java里新增加一个方法来实现这一功能。新增的方法如例8-16所示，就放在tracksNoLongerThan（）方法之后。

例8-16：根据艺人关联来过滤曲目

```
/**
 *Retrieve any tracks associated with artists whose name matches
 a SQL
 *string pattern.
 *
```

```
    *@param namePattern the pattern which an artist's name must
match
    *@param session the Hibernate session that can retrieve data.
    *@return a list of{@link Track}s meeting the artist name
restriction.
    */
    public static List tracksWithArtistLike (String namePattern,
Session session)
    {
        Criteria criteria=session.createCriteria (Track.class) ;
        Criteria artistCriteria=criteria.createCriteria ("artists") ; ❶
        artistCriteria.add (Restrictions.like ("name", namePattern)) ; ❷
        artistCriteria.addOrder (Order.asc ("name") .ignoreCase () ) ; ❸
        return criteria.list () ;
    }
```

❶前面开始的代码看起来很熟悉，这一行接着再用曲目对象的 **artists** 属性，又创建了一个 **Criteria** 实例，附加到我们用于选择曲目的那个 **Criteria** 实例上。这意味着我们或者可以增加约束到 **criteria** 上（应用到 **Track** 自身的属性），或者加到 **artistCriteria** 上（应用到与曲目关联的 **Artist** 实体的属性）。

❷在这个例子中，我们只对艺人的信息感兴趣，所以我们将结果限制为与艺人相关联的曲目，这些曲目的艺人中至少要有有一个艺人的名称必须匹配指定的模式（再一次，通过将约束应用到两个 **Criteria** 实例上，我们就能够同时限制 **Track** 和 **Artist** 的属性）。

注意：最终，在本书的这一版本中，能够看到我原本期望已久的输出结果。

❸ 我们要求按艺人的名字排序。编写本书第1版时，与当时可以使用的只有尝试性质的Criteria API相比，这是Hibernate3带来的又一个改进。原来只能够对最外层的条件查询进行排序，但不能对为关联而创建的子条件查询进行排序。如果试图这么做，就会得到一个UnsupportedOperationException。

为了看到这一查询的执行结果，我们还得做个修改。修改main()方法，让它调用这个新查询，如例8-17所示。

例8-17：调用新的曲目艺人姓名查询

```
.....
//Ask for a session using the JDBC information we've configured
Session session=sessionFactory.openSession ();
try{
//Print tracks associated with an artist whose name ends with"n"
List tracks=tracksWithArtistLike ("%n", session);
for (ListIterator iter=tracks.listIterator ();
.....
```

现在执行ant qtest，可以得到例8-18所示的结果。

例8-18：与姓名以字母"n"结尾的艺人相关联的所有曲目

```
qtest:
[java]Track: "The World'99" (Pulp Victim, Ferry Corsten) 00: 07:
05,
from Digital Audio Stream
[java]Track: "Adagio for Strings (Ferry Corsten Remix) " (William
Orbit,
Samuel Barber, Ferry Corsten) 00: 06: 35, from Compact Disc
```

```
[java]Track: "Gravity's Angel" (Laurie Anderson) 00: 06: 06, from  
Compact  
Disc
```

发生了什么事

注意：你也可以为用到的关联建立别名，并在表达式中使用。这会让情形变得更复杂，但是有用。改天研究一下吧。

如果查看这三个曲目的艺人列表，就会发现每个曲目的艺人姓名中至少有一个以"**n**"结尾，符合我们的要求。另外注意，我们也可以访问与曲目相关联的所有艺人，而并非只限于匹配**name**条件的艺人。这正是预期的效果，也是想要的效果，因为我们已经检索回实际的**Track**实体。你可以调用**setResultTransformer**

（**Criteria.ALIAS_TO_ENTITY_MAP**）方法来以不同的模式执行条件查询，让它返回一个层次式的**Map**对象的列表，条件查询会将结果过滤到这个**Map**列表的每个层次中。这部分主题已经超出本书讨论的范围，但是在**Hibernate**参考手册和**API**文档中提供了一些示例。

如果供你取回对象的数据表可能含有重复的实体，可以调用**Criteria**对象的**setResultTransformer**

（**Criteria.DISTINCT_ROOT_ENTITY**）方法，以达到与**SQL**语句的"**select distinct**"相同的效果。

示例查询

如果你觉得建立表达式和查询条件麻烦，而你又有对象能展示出要寻找的目标，就能够以它为示例，让Hibernate为你建立查询条件。

应该怎么做

我们需要在QueryTest.java里新增加另一个查询方法，将例8-19的代码添加到其他查询所在类的顶端。

例8-19：使用示例实体来生成条件查询

```
/**
 *Retrieve any tracks that were obtained from a particular source
media
 *type.
 *
 *@param media the media type of interest.
 *@param session the Hibernate session that can retrieve data.
 *@return a list of{@link Track}s meeting the media restriction.
 */
public static List tracksFromMedia (SourceMedia media, Session
session) {
    Track track=new Track () ; ❶
    track.setSourceMedia (media) ;
    Example example=Example.create (track) ; ❷
    Criteria criteria=session.createCriteria (Track.class) ; ❸
    criteria.add (example) ;
    criteria.addOrder (Order.asc ("title" ) ) ;
    return criteria.list () ;
}
```

❶我们先创建示例用的Track实例，再设置sourceMedia属性，以表示我们要查询的内容。

❷接着将它包装到一个Example对象中。这个对象可以在一定程度上让你控制在构建条件查询时将会用什么属性，以及如何匹配字符串。默认的行为是，值为null的属性将被忽略，对字符串值按照区分大小写的、逐字的方式进行比较。如果想在比较时忽略值为0的属性，可以调用example的excludeZeroes () 方法；或者，如果希望对值为null的属性进行匹配，可以调用excludeNone () 。而excludeProperty () 方法则可以让你明确地忽略指定名称的特定属性，不过这很像是在手工构建条件查询。为了调整字符串处理，还可以使用ignoreCase () 和enableLike () 方法，从它们的名字可以知道各自的用途。

❸接着，我们创建一个条件查询，就像本章的其他例子一样，不同的是这里为条件查询增加的是example，而不是使用Restrictions来创建Criterion。Hibernate会负责将example转换成相应的criteria。其他代码行与我们前面的查询方法类似：建立排序序列，运行查询，返回匹配的实体列表。

同样地，为了调用新的查询方法，我们也必须修改main () 方法。把来自CD的曲目都找出来吧。修改之处如例8-20所示。

例8-20：修改main () 方法，以调用示例驱动查询方法

```
.....
//Ask for a session using the JDBC information we've configured
Session session=sessionFactory.openSession () ;
try{
//Print tracks that came from CDs
List tracks=tracksFromMedia (SourceMedia.CD, session) ;
for (ListIterator iter=tracks.listIterator () ;
.....
```

运行这个版本的示例，产生的输出如例8-21所示。

例8-21： 以示例查询来自CD的曲目的结果

```
[java]Track: "Adagio for Strings (ATB Remix) " (ATB, Samuel
Barber,
William Orbit) 00: 07: 39, from Compact Disc
[java]Track: "Adagio for Strings (Ferry Corsten Remix) " (Samuel
Barber, William Orbit, Ferry Corsten) 00: 06: 35, from Compact
Disc
[java]Track: "Gravity's Angel" (Laurie Anderson) 00: 06: 06, from
Compact Disc
[java]Track: "Russian Trance" (PPK) 00: 03: 30, from Compact Disc
```

你可能觉得这个例子有些做作，因为我们并没有合适的Track对象可以作为示例，因此得在这个方法中创建一个示例。嗯，也许吧，但是采用此方法有个很有价值的理由：与纯条件查询相比，这种方法在编译期间会做更多的检查。虽然条件查询可以避免HQL查询在运行时的语法结构错误，但你还是可能会弄错属性的名称。而这些错误是编译器无法捕获的，因为名称只是字符串而已。当你建立示例查询时，实际上是以该实体的存取器方法（[\[1\]](#)）（mutator method）来设置属

性值的内容。这意味着，如果你输入错字，Java会在编译期间就捕获到它。

你大概想得到，也可以让子条件查询（`subcriteria`）使用示例来查询相关联的对象。我们可以重写`tracksWithArtistLike`（），使用一个作为示例用的`Artist`对象，而非自己手工建立查询条件。最后再调用示例对象的`enableLike`（）方法。例8-22演示了完成这一功能的简洁方法。

例8-22：更新艺人姓名查询，使用一个艺人对象作为示例

```
public static List tracksWithArtistLike (String namePattern,
Session session)
{
    Criteria criteria=session.createCriteria (Track.class) ;
    Example example=Example.create (new Artist (namePattern, null,
null)) ;
    criteria.createCriteria ("artists") .add (example.enableLike
()) .addOrder (
    Order.asc ("name") .ignoreCase () ) ;
    return criteria.list () ;
}
```

这一过程的输出与我们前面“手工”构建的内部条件查询生成的输出完全一样。如果你想运行这个例子，记得把`main`（）换回例8-17的样子。

注意：条件查询相当简单，对吧？和原来Hibernate 2中的相比，现在它们的功能更强大，我更喜欢它们了。

各种各样的查询可以增强用户界面的功能，数据驱动（data-driven）的Java应用程序的典型操作也可以表示成条件查询。在可读性、编译期类型检查甚至是代码紧凑性（令人惊讶）等各方面，条件查询都有其优点。就目前这些新的数据库API而言，我认为条件查询是赢家。

[1] 存取器方法指对象提供的用于访问其实例变量接口的方法。用于返回实例变量值的存取器方法称为获取方法（即get方法）；用于为实例变量赋值的存取器方法称为设置方法（即set方法）。

面向属性的Criteria工厂

我们已经看到，使用**Criteria API**来表达你想要实现的查询，通常都会有多种方法，到底使用哪一种方法取决于你对系统风格的偏好或考虑问题的方式。**Property**类提供了另一组替换方法，你应该对此有所了解。我们在这里不会深入研究这个类，因为它只是建立条件查询的另一种方法。不过，解释一下它的工作原理，也很重要，这样才不至于在运行许多示例时感到困惑，也不至于在浏览**Hibernate**高密度的**JavaDoc**时感觉有阻碍。（坦率地说，经过一两个示例以后，你肯定对以后会采用什么方法而有足够的想法了。）

Property是条件查询的另一个工厂类，与**Restrictions**非常像，本章已经用过了**Restrictions**（还有**Order**和**Projection**，都差不多）。其实，只使用**Property**就完全可以创建其他工厂类提供的所有查询约束条件。不像前面那样是先从感兴趣的约束条件开始，再为需要使用的属性命名，**Property**是先从属性开始，再挑选一个合适的约束条件。

注意：相当抽象！演示些例子吧！

和以前一样，先在想要查询的对象上创建一个**Criteria**。但接下来不是用以前的方法，例如：

```
criteria.add (Restrictions.le ("playTime", length) ) ;
```

而是用:

```
criteria.add (Property.forName ("playTime") .le (length) ) ;
```

这两条语句看起来非常像（只是侧重点稍微有些不同），就像用英语来表达同一个概念时也有多种方法一样。Property类也提供了很多方法，用于添加约束条件、排序以及投影。不能使用new（）方法来构造一个Property实例，你需要使用它的forName（）静态工厂方法，或是使用一个现有的Property实例，再调用它的getProperty（）方法，将其转换为它的某个组成属性。

以下列举一些更复杂的例子，以演示这种方法适用的场合。原来我们使用过以下语句:

```
criteria.addOrder (Order.asc ("name") .ignoreCase () ) ;
```

现在可以使用以下语句:

```
criteria.addOrder (Property.forName ("name") .asc () .ignoreCase  
() ) ;
```

对于投影，原来用的方法是，例如:

```
criteria.setProjection (Projections.max ("playTime") ) ;
```

与之等价的方法可以表示成：

```
criteria.setProjection (Property.forName ("playTime") .max () ) ;
```

注意：这简直就是财富带来的困窘！

这样，你需要从众多选择中挑选一种实现方法。有时，你正在着手解决的问题，或是代码中其他部分的难点，将决定使用某种风格的实现方式。或者，也许你只是更喜欢一种方法，就一直坚持使用这种方法了。但是，至少你现在知道了可能会遇到两种表达方法，应该能够理解任何一种条件表达式。附录B对所有这些工厂方法进行了总结。

选择太多了吗？没有？嗯，如果条件查询还不能很好的解决你的问题，或者你希望有一种更加强大的方法来取代本章介绍的所有选择（从本质来说，如果你更喜欢SQL的简洁），你就可以使用HQL提供的更为完整的功能。我们在下一章就会研究HQL。

其他

在条件查询中可以悄悄用点SQL，以便可以利用些数据库特定的功能或DBA技巧？使用子查询，或是通过离线（detached）查询，以便使你在一个Hibernate session之外创建一个查询？当执行查询时，可

以插入自己的Java代码来过滤结果？所有这些主题，可能还有更多其他的，已经超出本书讨论的范围。如果你准备学习它们，《Java Persistence with Hibernate》（[\[1\]](#)）这本书的"Advanced query options"（高级查询选项）一章就提供了一个好的概览，Hibernate JavaDoc和源代码也是不错的参考。

[\[1\]](#) 中文书名叫《Hibernate实战》。

第9章 浅谈HQL

前面几章已经多次用过HQL查询了。这里值得花点时间来了解HQL和SQL的差异，看看能用HQL做些什么有用的事。和本书其他章节一样，我们旨在提供有用的简介和示例，而不是完整的参考手册。

在第7章曾经提到，JPA的查询语言是HQL的一个子集。所以，如果学会了HQL，也应该能够读懂JPA查询语言（QL），如果，你用JPA编写查询，那么你就有足够的理由去超越它。大多数情况下，只要付出很小的努力，就可以学会如何使用Hibernate了。

正如第7章结尾所述，本章的示例将使用XML映射文件。所以，如果你在前面因为处理标注而修改了什么东西，在开始学习本章以前，最好先下载示例代码，并放到本章的目录中。

编写HQL查询

我们已经演示过，HQL查询中用到的语法成分会比SQL的少点（比如在第3章中，我们使用的查询语句通常就省略了"select"子句）。事实上，惟一真正需要指定的内容就是你感兴趣的类。例9-1显示了一个最简单的查询，对于取得数据库中所有持久保存的Track实例来说，这完全是一种有效的方式。

注意：HQL代表Hibernate Query Language。那SQL呢？这得看你问的是谁了。

例9-1：最简单的HQL查询

```
from Track
```

没什么内容，对吧？这一HQL就相当于例8-1，在那个例子中我们对Track类建立了一个条件查询，但没有提供任何查询条件。

默认情况下，Hibernate会自动“导入”（import）你映射的每个类的名称，这意味着当你想使用一个类时，不必提供完整的包名称，只需要简单的类名就可以了。只要映射的类名称是惟一的，就不需要在查询中使用完整限定的类名（fully qualified class name）。如果你喜欢，当然也可以这么做，本书就是这样做的，目的是帮助读者记住查询是按照Java数据bean及其属性来表示的，而并非数据表和字段（在SQL查询中使用这些）。例9-2的结果和第一个查询的结果完全相同。

例9-2：显式指定包名，但依然相当简单

```
from com.oreilly.hh.data.Track
```

如果需要映射的多个类具有相同的名称，你可以用完全限定的包名来引用每个类，也可以在其映射文件中使用import标签为该类或多

个类指定替换名称。还可以在映射文件中为最顶层的hibernate-mapping标签属性添加一个auto-import=false设置，以关掉自动导入（auto-import）的功能。

你可能已经习惯编写查询语句时不区分大小写，因为SQL的行为就是如此。大多数时候，HQL也是如此。但是类和属性的名称显然是例外。和Java的其他组成部分一样，它们是区分大小写的，所以你得把大小写弄对。

让我们看一个极端的示例，它把HQL的多态（polymorphic）查询能力推向它的逻辑上的极限，以此来了解HQL和SQL究竟在哪不同。

应该怎么做

突出SQL和HQL基本差别的最有效方式就是，考虑当以"from java.lang.Object"来进行查询时会发生什么事。乍一看，可能看不出什么意义！事实上，Hibernate能够支持返回多态结果的查询。如果映射到的类之间是彼此继承的，或者有共同的基类或接口，那么无论是否将这些类映射到相同的数据表或不同的数据表，都可以查询到它们的超类（superclass）。由于每个Java对象都继承自Object，所以这样的查询就等于要求Hibernate返回数据库里的每个实体。我们可以对查询测试做个快速的修改，以测试一下这一查询。把QueryTest.java复制成

QueryTest3.java，按例9-3所示的内容对它进行修改（大多数修改都是将这里不需要的示例查询删除掉，所以例9-3中看不到那些被删除的部分）。

如果你正在使用Eclipse，那么可以直接跳到第11章，阅读有关Hibernate Tools的章节，再回来继续学习本章的内容。那一章介绍的HQL Editor是一个非常方便的工具，可以帮助你处理我们需要查看的各种查询。用它来做实验太有效了，可以接合你自己的修改，帮助你更深入地学习HQL。

例9-3： 看吧，再也不用SQL了

```
package com.oreilly.hh;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import org.hibernate.criterion.*;
import com.oreilly.hh.data.*;
import java.util.*;
/**
 *Retrieve all persistent objects
 */
public class QueryTest3{
/**
 *Look up and print all entities when invoked from the command
line.
 */
public static void main (String args[]) throws Exception{
//Create a configuration based on the properties file we've put
//in the standard place.
Configuration config=new Configuration ();
config.configure ();
//Get the session factory we can use for persistence
SessionFactory sessionFactory=config.buildSessionFactory ();
//Ask for a session using the JDBC information we've configured
Session session=sessionFactory.openSession ();
```

```
try{
//Print every mapped object in the database
List all=session.createQuery ("from java.lang.Object") .list () ;
❶
for (Object obj: all) {
System.out.println (obj) ; ❷
}
}finally{
//No matter what, close the session
session.close () ;
}
//Clean up after ourselves
sessionFactory.close () ;
}
}
```

❶这行简单的代码就调用了奇妙而功能强大的HQL查询。

❷接下来所有需要做的就是循环遍历和打印输出我们找到的结果。除了调用取回的对象的`toString ()`方法以外，我们什么也不能做，因为我们不知道它们是什么类。作为共享接口，`Object`没有提供非常深入的方法。

在运行这个示例之前，还得再多做一件事。在`build.xml`中添加另一个构建目标，如例9-4所示。

和以前一样，假设你已经按照前面章节的说明搭建好了环境。也可以直接下载使用为这一章提供的示例源代码。如果你从第7章过来，将映射文档换成了标注，则要么你需要回到第6章的源代码，要么重新下载本章的源代码，因为本章的这些示例需要依靠映射文件的功能。

例9-4：调用这个特殊查询

```
<target name="qtest3" description="Retrieve all mapped objects"
depends="compile">
<java classname="com.oreilly.hh.QueryTest3" fork="true">
<classpath refid="project.class.path"/>
</java>
</target>
```

为了确保创建好所有的样例数据，先要运行`ant schema ctest atest`命令。接着，再执行`ant qtest3`命令以测试新的查询，得到的结果如例9-5所示。

例9-5: Hibernate能找到一切事物

```
%ant qtest3
Buildfile: build.xml
prepare:
usertypes:
codegen:
[hibernatetool]Executing Hibernate Tool with a Standard
Configuration
[hibernatetool]1.task: hbm2java (Generates a set of .java files)
compile:
[javac]Compiling 4 source files
to/Users/jim/Documents/Work/OReilly
/svn_hibernate/current/examples/ch09/classes
qtest3:
[java]com.oreilly.hh.data.Artist@8a137c[name='PPK'actualArtist='
null']
[java]com.oreilly.hh.data.Artist@79e4c[name='The
Buggles'actualArtist='n
ull']
[java]com.oreilly.hh.data.Artist@29ae5e[name='Laurie
Anderson'actualArti
st='null']
[java]com.oreilly.hh.data.Artist@76a9b6[name='William
Orbit'actualArtist
='null']
[java]com.oreilly.hh.data.Artist@801919[name='Ferry
Corsten'actualArtist
='null']
```

```

[com.oreilly.hh.data.Artist@efe4ac[name='Samuel
Barber'actualArtist
='null']]
[com.oreilly.hh.data.Artist@8e13ab[name='ATB'actualArtist='
null']]
[com.oreilly.hh.data.Artist@ad44f6[name='Pulp
Victim'actualArtist
='null']]
[com.oreilly.hh.data.Artist@8aaed5[name='Martin
L.Gore'actualArtist
='null']]
[com.oreilly.hh.data.Album@a1644b[title='Counterfeit
e.p.'tracks=[
com.oreilly.hh.data.AlbumTrack@132f26[track='com.oreilly.hh.data
.Track@d8f246[
title='Compulsion'volume='Volume[left=100,
right=100]'sourceMedia='CD']'], c
om.oreilly.hh.data.AlbumTrack@5ae101[track='com.oreilly.hh.data.
Track@9eab7[ti
tle='In a Manner of Speaking'volume='Volume[left=100,
right=100]'sourceMedia='
CD']'],
com.oreilly.hh.data.AlbumTrack@6a16ae[track='com.oreilly.hh.data.Tr
ac
k@10cf62[title='Smile in the Crowd'volume='Volume[left=100,
right=100]'source
Media='CD']'],
com.oreilly.hh.data.AlbumTrack@f7309a[track='com.oreilly.hh.da
ta.Track@9f332b[title='Gone'volume='Volume[left=100,
right=100]'sourceMedia='
CD']'],
com.oreilly.hh.data.AlbumTrack@97cf78[track='com.oreilly.hh.data.Tr
ac
k@d86cae[title='Never Turn Your Back on Mother
Earth'volume='Volume[left=100,
right=100]'sourceMedia='CD']'],
com.oreilly.hh.data.AlbumTrack@b5d53a[track=
'com.oreilly.hh.data.Track@c74b55[title='Motherless
Child'volume='Volume[left=
100, right=100]'sourceMedia='CD']']]]]
[com.oreilly.hh.data.Track@e74d83[title='Russian
Trance'volume='Vol
ume[left=100, right=100]'sourceMedia='CD']]
[com.oreilly.hh.data.Track@5d8362[title='Video Killed the
Radio Star
'volume='Volume[left=100, right=100]'sourceMedia='VHS']]
[com.oreilly.hh.data.Track@5c9ab5[title='Gravity's
Angel'volume='Vo

```

```

    lume[left=100, right=100]'sourceMedia='CD']
    [java]com.oreilly.hh.data.Track@b0e76a[title='Adagio for Strings
(Ferry C
    orsten Remix)'volume='Volume[left=100,
right=100]'sourceMedia='CD']
    [java]com.oreilly.hh.data.Track@28ea3f[title='Adagio for Strings
(ATB Rem
    ix)'volume='Volume[left=100, right=100]'sourceMedia='CD']
    [java]com.oreilly.hh.data.Track@2af08b[title='The
World'99'volume='Volu
    me[left=100, right=100]'sourceMedia='STREAM']
    [java]com.oreilly.hh.data.Track@164feb[title='Test Tone
1'volume='Volume
    [left=50, right=75]'sourceMedia='null']
    [java]com.oreilly.hh.data.Track@d8f246[title='Compulsion'volume=
'Volume[
    left=100, right=100]'sourceMedia='CD']
    [java]com.oreilly.hh.data.Track@9eab7[title='In a Manner of
Speaking'vol
    ume='Volume[left=100, right=100]'sourceMedia='CD']
    [java]com.oreilly.hh.data.Track@10cf62[title='Smile in the
Crowd'volume=
    'Volume[left=100, right=100]'sourceMedia='CD']
    [java]com.oreilly.hh.data.Track@9f332b[title='Gone'volume='Volum
e[left=1
    00, right=100]'sourceMedia='CD']
    [java]com.oreilly.hh.data.Track@d86cae[title='Never Turn Your
Back on Mot
    her Earth'volume='Volume[left=100, right=100]'sourceMedia='CD']
    [java]com.oreilly.hh.data.Track@c74b55[title='Motherless
Child'volume='V
    olume[left=100, right=100]'sourceMedia='CD']
BUILD SUCCESSFUL
Total time: 9 seconds

```

发生了什么事

嗯，仔细想一想，这可真不简单，因为Hibernate得分别做好几次SQL查询才能获得我们想要的所有结果。Hibernate在幕后做了很多工作，才能把每个它知道的持久化实体的列表交给我们。虽然很难想象

实际需要这么做的情况，但这确实突出了HQL和Hibernate有趣的能力所在。

有时候某些查询确实有些难以理解，但却非常有用。所以记住这一点很有价值：跨表的多态查询（polymorphic query）不仅可行，而且容易使用。

当你执行需要多条SQL查询才能完成的HQL查询时，会有些限制需要注意。不能使用order by子句对整个结果集进行排序，也不能使用Query接口scroll（）方法遍历整个结果集。

其他

关联（association）和连接（join）呢？这些也容易处理。只要使用点号作为分隔符（delimiter），顺着属性链走，就能访问关联对象。为了帮助你引用查询表达式中的特定实体，HQL可以让你指定别名（就像SQL一样）。如果你想引用同一个类的两个不同的实体，别名就特别重要，例如：

```
from com.oreilly.hh.Track as track1
```

相当于：

```
from com.oreilly.hh.Track track1
```

采用哪一种方法取决于你的习惯，或者取决于你为项目准备的编码风格指南。

我们对其他HQL元素进行了足够的介绍，就是想让它们吸引起你的兴趣。稍后我们会看看一些连接的示例。

选择属性和其他部件

到目前为止，我们用的查询返回的都是整个持久化对象。这是像Hibernate这类对象/关系映射服务组件最常见的用法，所以应该没什么可奇怪的。在得到这些持久化对象以后，就可以在你熟悉的Java代码世界中用它们做任何你想做的事。但是在有些情况下，你可能只想取出组成对象的所有属性的一个子集，例如生成报表。HQL也能满足这样的需要，和普通SQL的使用方式一样，也就是在select子句中使用SQL投影（SQL-projection）。

应该怎么做

假设我们想在QueryTest.java的基础上进行修改，使其只显示符合查询条件的曲目的标题，而且想从数据库中一开始提取的信息就只包含曲目的标题。首先，要修改例3-11的查询，让它只检索回title属性。编辑Track.hbm.xml，使其查询内容看起来如例9-6所示。

例9-6：只获取短曲目的标题

```
<query name="com.oreilly.hh.tracksNoLongerThan">
<![CDATA[
select track.title from com.oreilly.hh.Track as track
where track.playTime<=: length
]]>
</query>
```

要确保让QueryTest.java中的tracksNoLongerThan () 方法使用这个查询。（如果你在第8章将它修改成使用条件查询，就要再改回例3-12的样子。为了节省查找的麻烦，例9-7又重新生成了一次。）

例9-7: HQL驱动查询方法，使用例9-6中映射的查询

```
public static List tracksNoLongerThan (Time length, Session
session) {
    Query query=session.getNamedQuery (
        "com.oreilly.hh.tracksNoLongerThan");
    query.setTime ("length", length);
    return query.list ();
}
```

最后，还需要更新main () 方法（如例9-8所示），以反映一个事实：这个查询方法现在返回的是title属性，而不是整个Track记录。这个属性的类型是String，所以，现在这个查询方法返回的是一个由String组成的List对象。

例9-8: 修改QueryTest的main () 方法，以处理标题查询

```
//Print the titles of tracks that will fit in five minutes
List titles=tracksNoLongerThan (Time.valueOf ("00: 05: 00"),
session);
for (ListIterator iter=titles.listIterator ();
    iter.hasNext (); ) {
    String title= (String) iter.next ();
    System.out.println ("Track: "+title);
}
```

这些修改都相当简单，而Java程序中查询返回的类型和List元素之间的关系也一目了然。使用`ant qtest`命令运行这个例子，就会得到类似例9-9的输出结果，至于实际显示的数据，则取决于你已经设定的数据。（如果还没有任何数据，或者你想要的结果，输出前可以执行`ant schema ctest atest qtest`，这样会重建测试数据。）

例9-9：只列出时间长度不超过5分钟的曲目的标题

```
qtest:
[java]Track: Russian Trance
[java]Track: Video Killed the Radio Star
[java]Track: Test Tone 1
[java]Track: In a Manner of Speaking
[java]Track: Gone
[java]Track: Never Turn Your Back on Mother Earth
[java]Track: Motherless Child
```

其他

要返回多个属性？当然可以。如果你使用连接（`join`）或者你的查询对象中有多个组件或关联（毕竟这是面向对象关联非常方便的方式），在这些情况下，属性都可以来自多个对象。如同SQL那样，你所做的就是列出需要的属性，并以逗号分隔。举一个简单的例子，假设我们要得到曲目的标题和ID。调整`Track.hbm.xml`，使查询内容如例9-10所示。

例9-10：从一个对象中选择多个属性

```
<query name="com.oreilly.hh.tracksNoLongerThan">
<![CDATA[
select track.id, track.title from com.oreilly.hh.Track as track
where track.playTime<=: length
]]>
</query>
```

根本不需要修改查询方法，还是以相同的名称调用这个查询，并传递相同的命名参数，再返回保存有结果的列表。但是这个列表中现在包含什么？我们需要更新`main()`中的循环，才能同时显示曲目的ID和标题。

像这种情况，查询结果的每一“行”（`row`）都要返回多个值，所以Hibernate返回的List对象中的每个条目都会包含一个对象数组。每个数组都包含我们所选择的属性，排列顺序按照各属性在查询语句中的位置而定。这样，我们会得到一个包含两个元素的数组的列表，每个数组内包含一个Integer对象，再接着包含一个String对象。

例9-11演示了如何更新QueryTest.java中的`main()`，以处理这些数组。

例9-11：处理查询结果中多个单独的属性

```
//Print IDs and titles of tracks that will fit in five minutes
List titles=tracksNoLongerThan (Time.valueOf ("00: 05: 00"),
session);
for (ListIterator iter=titles.listIterator ();
iter.hasNext (); ) {
Object[]row= (Object[]) iter.next ();
Integer id= (Integer) row[0];
```

```
String title= (String) row[1];
System.out.println ("Track: "+aTitle+"[ID="+id+'']');
}
```

经过这些修改以后，执行`ant qtest`，可以得到类似例9-12的输出。

例9-12：列出曲目的标题和ID

```
qtest:
[java]Track: Russian Trance[ID=1]
[java]Track: Video Killed the Radio Star[ID=2]
[java]Track: Test Tone 1[ID=7]
[java]Track: In a Manner of Speaking[ID=9]
[java]Track: Gone[ID=11]
[java]Track: Never Turn Your Back on Mother Earth[ID=12]
[java]Track: Motherless Child[ID=13]
```

我希望你在看这个示例时会想：“这种使用Track对象属性的方法真差！”。如果你没这么想，比较一下例9-11和例3-7中的对应代码。后者更为精简而自然，甚至能够显示出更多的曲目信息。如果要提取出映射对象的信息，最好是充分利用映射的能力以提取出对象的真正的实例，这样你就能用表达能力更强而且类型安全的Java代码来处理它的属性。

注意：这是那种残酷的玩笑吗？

既然如此，为什么还要这样做？嗯，在某些情况下，用HQL检索回多个值有其用途所在。例如，对某些映射类，你只想从每个类中取出一个属性；或者，你可能想通过在select子句中列出一些类名来返回

一组相关的类。对于这些情况而言，知道这种技巧有其价值所在。如果你的映射对象有很多大型的（或非延迟加载关联的）属性，但你只对其中的一两个属性感兴趣，这一技巧也许能在性能上让你受益匪浅。

此外，当你从不同映射对象中选择一大组属性以建立报表时，还有另一种技巧可以用来方便地构建良好的对象结构。**HQL**可以让你在**select**子句内构建并返回任意对象。所以，你可以创建一个专门的报表类，它的属性就是你的报表需要的值，然后在查询中返回这个报表类的实例，而不是返回晦涩难懂的**Object**数组。如果我们定义一个具有**id**和**title**属性的**TrackSummary**类，以及与之相配的构造函数，如此一来就能够使用以下的查询：

```
select new TrackSummary (track.id, track.title)
```

而不是使用：

```
select track.id, track.title
```

这样，我们就不用在处理查询结果的代码中对数组进行处理。（再一次，就这个例子来说，简单地返回整个**Track**实例是比较有意义的，但是当你要使用来自多个对象的属性或者像聚合函数（**aggregate function**）这类合成结果时（后面会有演示），这样做就有用处了。）

排序

可以使用SQL风格的"order by"子句控制结果输出的顺序，这应该不会令你感到惊讶。前面几章我们曾经简单提到过这一点，其工作原理正如你所料想的那样。可以使用返回对象的任何属性来建立排序，也可以使用多个属性来建立子排序（**subsort**），以便在第一个属性值相同时能够继续以其他属性值对其结果进行排序。

应该怎么做

排序非常简单：列出想要用来对查询结果进行排序的值。和往常一样，SQL使用数据库表的字段，而HQL使用对象的属性。对于例9-13来说，我们更新了例9-10的查询，让它以倒序的字母顺序显示结果。

注意：和SQL一样，递增的顺序使用"asc"，而递减的顺序使用"desc"。

例9-13：在Track.hbm.xml中加点新东西，按标题倒序对结果进行排序

```
<query name="com.oreilly.hh.tracksNoLongerThan">
<![CDATA[
```



```
select track.id, track.title from com.oreilly.hh.Track as track
where track.playTime<=: length
order by track.title desc
]]>
</query>
```

运行这个示例的输出结果如你所料（如例9-14所示）。

例9-14：以倒序的字母顺序列出的标题和ID

```
%ant qtest
Buildfile: build.xml
prepare:
[copy]Copying 1 file
to/Users/jim/Documents/Work/OReilly/svn_hibernate
/current/examples/ch09/classes
usertypes:
.....
qtest:
[java]Track: Video Killed the Radio Star[ID=2]
[java]Track: Test Tone 1[ID=7]
[java]Track: Russian Trance[ID=1]
[java]Track: Never Turn Your Back on Mother Earth[ID=12]
[java]Track: Motherless Child[ID=13]
[java]Track: In a Manner of Speaking[ID=9]
[java]Track: Gone[ID=11]
BUILD SUCCESSFUL
Total time: 9 seconds
```

使用聚合值

我们时常需要从数据库中提取摘要信息，尤其是在做报表的时候：有多少？平均值？最长的？HQL也支持这一功能，办法就是提供与SQL类似的聚合函数（aggregate function）。当然，在HQL中，这些函数是针对持久化类的属性而进行运算的。

应该怎么做

我们以目前现有的一些查询测试框架为基础试一试吧。首先，在Track.hbm.xml中现有的查询内容之后再新添加例9-15所示的查询内容。

例9-15：一个用于收集曲目的聚合信息的查询

```
<query name="com.oreilly.hh.trackSummary">
<![CDATA[
select count (*), min (track.playTime), max (track.playTime)
from Track as track
]]>
</query>
```

我原来试图取出平均播放时间，但很不幸，HSQLDB不知道应该如何为非数值的字段计算平均值，因为这个属性是保存在数据库类型为date（日期）的字段中。

接下来，我们必须写一个方法去执行这个查询，并显示结果。把例9-16所示的代码加进QueryTest.java，就放在tracksNoLongerThan（）方法的后面。

例9-16：执行trackSummary查询的方法

```
/**
 *Print summary information about all tracks.
 *
 *@param session the Hibernate session that can retrieve data.
 */
public static void printTrackSummary (Session session) {
    Query query=session.getNamedQuery
("com.oreilly.hh.trackSummary");
    Object[]results= (Object[]) query.uniqueResult ();
    System.out.println ("Summary information: ");
    System.out.println ("Total tracks: "+results[0]);
    System.out.println ("Shortest track: "+results[1]);
    System.out.println ("Longest track: "+results[2]);
}
```

由于我们只在查询中使用聚合函数，因此我们知道返回的结果只会有一行数据。在这种情况下，就可以使用Query接口提供的另一个更加方便的uniqueResult（）方法，这样能免去取回一个列表，并把列表的第一个元素取出来的麻烦。如本节前面的9.2节所述，因为我们要求的是几个不同的值，因此结果将会是一个Object数组，其元素就是我们请求的那些值，它们的顺序与查询结果中的排列顺序相同。（这段代码看起来有些难看，使用了晦涩难懂的数字式数组引用。这就是我们演示的为什么需要为查询创建“报表对象”的原因了，使用报表对象将会让处理变得比现在要方便得多。在普通的SQL语句中，你可以为

字段起个名称，通过名称从ResultSet（结果集）中取回字段值；在HQL中，你可以声明类，并在查询中创建这个类的实例。）

我们还需要在main（）中新添加一行代码来调用这个方法。可以将这行代码放在打印输出选择的曲目详细信息的循环之后，如例9-17所示。

例9-17：在QueryTest.java中的main（）中新增加一行，以显示新的摘要信息

```
.....
System.out.println ("Track: "+aTitle+"[ID="+anID+']') ;
}printTrackSummary (session) ;
}finally{
//No matter what, close the session
.....
```

增加好代码以后，执行ant qtest，就可以得到新的输出，如例9-18所示。

例9-18：摘要输出

```
.....
qtest:
[java]Track: Video Killed the Radio Star[ID=2]
[java]Track: Test Tone 1[ID=7]
[java]Track: Russian Trance[ID=1]
[java]Track: Never Turn Your Back on Mother Earth[ID=12]
[java]Track: Motherless Child[ID=13]
[java]Track: In a Manner of Speaking[ID=9]
[java]Track: Gone[ID=11]
[java]Summary information:
```

```
[java]Total tracks: 13
[java]Shortest track: 00: 00: 10
[java]Longest track: 00: 07: 39
BUILD SUCCESSFUL
Total time: 9 seconds
```

相当简单吧。我们再试试比较难处理的——从连接表中获取信息。曲目有一个关联到艺人的集合。假设我们想取出和特定艺人相关联的曲目的摘要信息，而不是所有曲目的摘要信息。例9-19演示了查询语句中新增加的部分。

例9-19: 获取特定艺人相关的曲目摘要信息

```
<query name="com.oreilly.hh.trackSummary">
<![CDATA[
select count (*), min (track.playTime), max (track.playTime)
from Track as track
where: artist in elements (track.artists)
]]>
</query>
```

为了过滤想看到的曲目，我们新增了一个`where`子句，以及一个命名参数`artist`。Hibernate的`in`运算符还有另外一个用途，可以像在普通的SQL语句中那样来使用`in`运算符以给定某个属性可能取值的列表，也可以按这里采用的方式来使用`in`。这个语句是告诉Hibernate，我们感兴趣的曲目是其`artists`集合中含有特定艺人的那些曲目。为了调用这一查询，需要对`printTrackSummary ()`做点修改，如例9-20所示。

例9-20: 改进`printTrackSummary ()`，以处理特定艺人的曲目

```

/**
 *Print summary information about tracks associated with an
artist.
 *
 *@param artist the artist in whose tracks we're interested.
 *@param session the Hibernate session that can retrieve data.
 */
public static void printTrackSummary (Artist artist, Session
session) {
    Query query=session.getNamedQuery
("com.oreilly.hh.trackSummary");
    query.setParameter ("artist", artist);
    Object[]results= (Object[]) query.uniqueResult ();
    System.out.println ("Summary of tracks by"+artist.getName ()
+': ');
    System.out.println ("Total tracks: "+results[0]);
    System.out.println ("Shortest track: "+results[1]);
    System.out.println ("Longest track: "+results[2]);
}

```

没多加什么，对吧？最后，调用该方法的那行代码需要用 一个参数来指定一个艺人对象。我们可以再次使用 `CreateTest.java` 中方便的 `getArtist ()` 方法。修改 `QueryTest.java` 的 `main ()` 方法中对该方法的调用代码，使其如例9-21所示。

例9-21：调用已改进的 `print, TrackSummary ()`

```

.....
System.out.println ("Track: "+title+"[ID="+id+']');
}
printTrackSummary (CreateTest.getArtist ("Samuel Barber",
false, session), session);
}finally{
//No matter what, close the session
.....

```

现在，当你执行 `ant qtest` 时，就会看到类似例9-22的信息。

例9-22：运行摘要查询，以取得艺人Samuel Barber的所有曲目

```
qtest:
[java]Track: Video Killed the Radio Star[ID=2]
[java]Track: Test Tone 1[ID=7]
[java]Track: Russian Trance[ID=1]
[java]Track: Never Turn Your Back on Mother Earth[ID=12]
[java]Track: Motherless Child[ID=13]
[java]Track: In a Manner of Speaking[ID=9]
[java]Track: Gone[ID=11]
[java]Summary of tracks by Samuel Barber:
[java]Total tracks: 2
[java]Shortest track: 00: 06: 35
[java]Longest track: 00: 07: 39
```

发生了什么事

注意：试着以普通的SQL做类似的事！

这根本不费吹灰之力，因此值得花点时间来欣赏一下Hibernate为我们做了多少事。我们调用的`getArtist()`方法会返回与"Samuel Barber"对应的Artist实例，再将整个实例对象作为命名参数传递给HQL查询，而Hibernate也知道应该如何使用Artist对象的id属性和TRACK_ARTISTS表，把连接查询放在一起，以实现我们在例9-10中简单描述的那种复杂情况。

得到的结果反映出示例数据中那两个艺人相互混合的"Adagio for Strings"曲目。因为它们都超过了5分钟，所以没有出现在曲目列表中。

编写原生SQL查询

HQL的功能强大、使用方便，而且能和Java代码中的对象自然地结合起来，既然如此，有什么理由不去用HQL呢？嗯，项目所用的数据库可能有某些特殊功能是它原生（native）的SQL方言才支持的，这时就不能用HQL了。如果你愿意接受使用原生SQL会让以后数据库的更换变得更加困难的事实，**Hibernate**还是可以让你使用这种原生SQL方言来编写查询，同时仍然帮助你以对象属性的方式来编写表达式，并把查询结果转换成对象（当然，如果你不想依靠**Hibernate**的帮助，也可以使用原始的JDBC连接来执行普通的SQL查询）。

另一种可能不完全符合你的情况是，你正处于将当前现有的基于JDBC的项目移植到基于**Hibernate**的过程中，而你只是想先前进几小步，并不想立刻彻底重新改写每个查询。

应该怎么做

如果准备将查询语句的文本嵌入到Java源代码中，可以用Session类的createSQLQuery（）方法，而不是用例3-7的createQuery（）方法。当然，你应该知道有比这种编码方式更好的办法，所以我就不举例说明这种方法了。比较好的做法是把查询语句放在映射文档中，如

例3-11所示。区别在于，现在应该使用`<sql-query>`标签，而不是我们一直在用的`query`标签。你也需要告诉Hibernate要返回什么映射类，以及在查询中引用该类（及其属性）的别名是什么。

假设我们想知道所有恰好结束在半分钟的曲目（换句话说，自动唱片机系统显示的播放时间是`h: mm: 30`），不过，这个例子有点做作。简单的方法就是使用HSQLDB内建的`SECOND`函数，它会返回`Time`值中秒数的部分。由于HQL不知道HSQLDB SQL方言中这一特殊的函数，所以我们只得使用原生的SQL查询。例9-23演示了这个例子的实现，将它添加到`Track.hbm.xml`中的HQL查询之后。

例9-23：在Hibernate映射文档中嵌入一个原生的SQL方言查询

```
<sql-query name="com.oreilly.hh.tracksEndingAt">
<return alias="track"class="com.oreilly.hh.data.Track"/>
<![CDATA[
select{track.*}
from TRACK as{track}
where SECOND ({track}.PLAYTIME) =: seconds
]]>
</sql-query>
```

`return`标签是告诉Hibernate，我们准备在查询中使用`track`别名来引用`Track`对象，这样就能够在查询语句中使用简写的`{track.*}`来引用`TRACK`数据表所需要的所有字段以创建`Track`实例（注意，在查询语句中这样使用别名时必须用大括号将别名括起来。这能让我们“脱

离”原生的SQL语句环境，用Hibernate映射类和属性来描述查询内容）。

查询中的where子句使用了HSQLDB SECOND函数来过滤查询结果，只在结果中保留时间长度值的秒数部分具有特定值的曲目。幸好，即便我们构建了一个原生的SQL查询，依然能够利用Hibernate很棒的命名查询参数。在这个示例中，我们传递了一个名为"seconds"的值以控制查询返回的结果。（即便在SQL查询中，也不需要使用大括号来告诉Hibernate你正在使用命名参数。Hibernate的解析器足够智能，以致于可以自动明白你写的是什么。）

使用该映射SQL查询的代码和前面示例中使用HQL查询的代码并没有什么不同。可以用getNamedQuery（）方法来加载这两种查询，而且这两种查询都实现了Query接口。所以，调用该查询的Java方法看起来应该很熟悉。将例9-24中的代码添加到QueryTest.java中printTrackSummary（）方法的后面。

例9-24：调用原生的SQL映射查询

```
/**
 *Print tracks that end some number of seconds into their final
 minute.
 *
 *@param seconds, the number of seconds at which we want tracks
 to end.
 *@param session the Hibernate session that can retrieve data.
 **/
```

```
public static void printTracksEndingAt (int seconds, Session
session) {
    Query query=session.getNamedQuery
("com.oreilly.hh.tracksEndingAt");
    query.setInteger ("seconds", seconds);
    List results=query.list ();
    for (ListIterator iter=results.listIterator (); iter.hasNext
() ; ) {
        Track track= (Track) iter.next ();
        System.out.println ("Track: "+track.getTitle () +
", length="+track.getPlayTime () );
    }
}
```

最后，在`main ()`中加上几行代码以调用这个方法。例9-25演示了将其添加到对`printTrackSummary ()`的调用之后的样子。

例9-25: 调用`printTracksEndingAt ()`以显示终止于半分钟的曲目

```
.....
printTrackSummary (CreateTest.getArtist ("Samuel Barber",
false, session), session);
System.out.println ("Tracks ending halfway through final
minute: ");
printTracksEndingAt (30, session);
}finally{
//No matter what, close the session
.....
```

执行`ant qtest`命令后，这些修改会产生额外的输出结果，如例9-26所示。

例9-26: 原生SQL查询的输出示范

```
qtest:
[java]Track: Video Killed the Radio Star[ID=2]
```

```
.....  
[java]Summary of tracks by Samuel Barber:  
[java]Total tracks: 2  
[java]Shortest track: 00: 06: 35  
[java]Longest track: 00: 07: 39  
[java]Tracks ending halfway through final minute:  
[java]Track: Russian Trance, length=00: 03: 30  
BUILD SUCCESSFUL  
Total time: 10 seconds
```

与HQL查询相比，使用原生SQL查询时需要注意更多不同层次上的细节，也更乏味（尤其是当你的查询变得复杂或涉及对多个数据表的引用时），但是，它偶尔会在某些场合真的可以派上用场，这也是一件好事。

其他

从Hibernate 3起，`sql-query`映射也成为在Hibernate中操纵数据库存储过程的入口点（如果这些与你没什么关系，不必担心；这只是意味着你现在还不需要为它们操心，当你需要使用它们时，就会知道是怎么回事了）。对于可以调用的查询的种类，以及查询可以返回的类型，都有一些明确的限制，而且这些限制会随着数据库的不同而有所不同。更详尽的细节，可以查看Hibernate参考手册（[\[1\]](#)）。

嗯，还有其他的東西嗎？你會懷疑本章幾乎沒有談到用HQL可以做些什麼。的確如此。當你開始結合這些功能，並運用集合、關聯以及功能強大的表达式時，就能完成一些令人刮目相看的事情。我們不

可能在这本介绍性的图书中涉及方方面面的细节，所以你得自己去阅读Hibernate参考手册中有关HQL的那一节及其示例，然后再自己做些实验。

当你阅读Hibernate参考手册中有关"Hibernate Query Language"（[\[2\]](#)）的那一章时，一定要看看表达式中能用到的那些有趣的事项，尤其是与集合有关的情况。别忘了你可以用数组括号表示法来选择集合中具有特定索引值的元素，甚至可以在括号中置入算术表达式。

Hibernate参考手册的HQL那一章在冗长的示例之后，接着的一节是“小技巧和小窍门”（Tips and Tricks），这一节给出了一些有用的建议，谈到了如何在不同数据库环境中有效地工作，以及使用各种Hibernate接口来达到你想都想不到的效果（如果你有SQL使用基础，就更是如此）。你也可以仔细研读本书附录E提到的那些图书。希望这里的讨论能够帮助你奠定好基础，并能作为日后探索研究的出发点、支柱以及动机！

注意：这不是你老爸那个时代的SQL.....

[\[1\]](#)

http://www.hibernate.org/hib_docs/v3/reference/en/html_single/#sp_query.

[\[2\]](#) http://www.hibernate.org/hib_docs/v3/reference/en/html/queryhql.html.

第二部分 与其他工具的集成

到目前为止，我们的示例只涉及除Hibernate以外非常有限的一组工具：Ant、Maven Ant Tasks、Hibernate Tools以及我们的关系数据库HSQLDB。为我们快速地“品尝”Hibernate并体验它的主要功能而提供一个平台，这些工具再合适不过了。不过，在现实世界中，你可能经常会需要以不同的方式来使用Hibernate，并与其他有用的工具包配合使用。

幸好，这些工具的用法还算简单！所以我们换个方向，开始介绍一些新面孔，比如流行的MySQL数据库和Eclipse IDE（集成开发环境）。接着我们将更深入地讨论Maven，而不是像原来那样，只是将它作为让示例可以正常工作的一块跳板，演示了一些比Ant Tasks更为有用的Maven用法。最后，我们将重点介绍Spring和Stripes，这是两个非常有用的开源项目，与Hibernate结合得相当完美。

第10章 将Hibernate连接到MySQL

建立MySQL数据库

虽然HSQLDB是一个漂亮的、完整的（self-contained）数据库，不过你的实际项目可能并不需要这种嵌入式的Java数据库。事实上，你更可能需要同某些现有的、外部的数据库打交道。幸好，这也没什么困难的（假设你已经安装好了数据库，并可以让它正常运行起来，不过，这些（安装和配置）内容肯定不在本书讨论的范围之内）。

注意：这个例子假设你已经有了一个可运行的MySQL实例，可以管理它。

为了突出Hibernate在数据库选择上的灵活性，我们先来看看如果想让Hibernate连接到MySQL数据库，应该怎么修改第2章中Hibernate的配置。

应该怎么做

连接到MySQL服务器，创建一个新的数据库，如例10-1所示。

例10-1：建立MySQL数据库notebook_db

```
%mysql-u root-p
Enter password:
Welcome to the MySQL monitor. Commands end with; or\g.
Your MySQL connection id is 3 to server version: 5.0.21
Type'help; 'or'\h'for help.Type'\c'to clear the buffer.
mysql>CREATE DATABASE notebook_db;
Query OK, 1 row affected (0.03 sec)
mysql>GRANT ALL ON
notebook_db.*TO'jim'@'janus.reseune.pvt'IDENTIFIED
BY's3cret';
```

```
Query OK, 0 rows affected (0.02 sec)
mysql>quit;
Bye
```

注意一下你创建的数据库的名称，以及授予能够访问它的用户名和口令，需要将这些信息输入到`hibernate.cfg.xml`配置文件中，如稍后的例10-3所示（在实际的数据库使用中，希望你使用比这个例子的口令更为健壮的口令）。如果你的数据库服务器与运行示例代码的计算机不是同一台机器，那么需要将`GRANT`命令行中的`localhost`替换为用于运行示例的主机名（或者，如果你是在一个相对安全的私有网络中，那么还可以使用“%”来表示容许接受任何主机的访问）。

连接到MySQL

接下来，我们需要用一个JDBC驱动程序才能够连接到MySQL。为了获取这个驱动程序，可以在build.xml中再添加一个依赖，如例10-2所示。对于几种不同的数据库，都有它们各自可供使用的驱动程序，这一点不错。这些不同的驱动程序不会冲突，因为Hibernate配置文件会指定最终使用哪个驱动程序（如果你对如何“手工”获取这种驱动程序感到好奇，那么可以从MySQL的官方网站（[\[1\]](#)）下载Connector/J）。不过，如果你想避免以后Hibernate配置文件的读者会感到这部分有些混乱，则可以放心地删除<hsqldb>依赖和<db>构建目标，因为以后不再需要用GUI（图形用户界面）查看HSQLDB中的数据了。

例10-2：在build.xml中添加对MySQL驱动程序的项目依赖

```
.....
<artifact: dependencies pathId="dependency.class.path">
  <dependency
groupId="hsqldb"artifactId="hsqldb"version="1.8.0.7"/>
  <dependency groupId="mysql"artifactId="mysql-connector-java"
version="5.0.5"/>
  <dependency groupId="org.hibernate"artifactId="hibernate"
version="3.2.5.ga">
.....
```

说到配置文件，现在就应该编辑hibernate.cfg.xml，以使用新的驱动程序和我们刚才创建的数据库了。例10-3演示了如何使用前面例10-1创建的数据来建立到我自己的MySQL数据库实例的连接。你需要根据你自己的数据库服务器、数据库以及你所选择的登录身份认证来调整这些数据库连接的配置值（如果你使用的是MM.MySQL，则应该使用旧版的MySQL JDBC驱动程序，相应的驱动程序类名称应该是com.mysql.jdbc.Driver）。

还需要删除配置文件底部的jdbc.batch_size属性。报告来自批处理内的实际问题，MySQL驱动程序没有任何问题；与进程外（out-of-process）数据库相比，对于前面使用的HSQLDB驱动程序，如果关闭这个批处理选项，则会对性能造成很大的不良影响。

例10-3：修改hibernate.cfg.xml以连接到新的MySQL数据库

```
<?xml version='1.0'encoding='utf-8'?>
<! DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
<session-factory>
<!--SQL dialect-->
<property name="dialect">org.hibernate.dialect.MySQL5Dialect
</property>
<!--Database connection settings-->
<property name="connection.driver_class">com.mysql.jdbc.Driver
</property>
<property
name="connection.url">jdbc: mysql: //localhost/notebook_db
</property>
<property name="connection.username">jim</property>
```

```
<property name="connection.password">s3cret</property>  
<property name="connection.shutdown">>true</property>  
.....
```

如果不是在运行测试示例的机器上运行MySQL，而是在另外的机器上运行MySQL，则第三个属性定义必须能够反映出你的数据库服务器的名称。另外，`connection.shutdown`配置属性也不再需要了，不过，放在那里也没什么影响。

[1] <http://www.mysql.com/products/connector/j/>.

尝试一下

一切准备好以后，再回到第2.3节建立的那个数据库模式创建的示例。这一次它应该在MySQL服务器上创建数据库模式，而不是在嵌入式HSQLDB中。数据库模式创建的输出如例10-4所示，不过，实际输出的内容要比这个例子列举的内容还要长，所以就删除了部分重复和繁琐的输出，以突出一些特别重要的片段，通过它们来证明我们的新设置已经生效了（注意，我们以一个单独的ant prepare调用作为开始，因为这是我们第一次在本章的示例目录运行程序，在Ant第一次开始运行schema构建目标时，我们需要将正确的文件放在类路径上）。

例10-4：连接MySQL并创建数据库模式

```
%ant prepare
Buildfile: build.xml
Downloading: mysql/mysql-connector-java/5.0.5/mysql-connector-
java-5.0.5.pom
Transferring 1K
Downloading: mysql/mysql-connector-java/5.0.5/mysql-connector-
java-5.0.5.jar
Transferring 500K
prepare:
[mkdir]Created
dir: /Users/jim/svn/oreilly/hibernate/current/examples
/ch10/classes
[copy]Copying 5 files to/Users/jim/svn/oreilly/hibernate/current
/examples/ch10/classes
BUILD SUCCESSFUL
Total time: 3 seconds
%ant schema
Buildfile: build.xml
```

```

prepare:
usertypes:
[javac]Compiling 2 source files
to/Users/jim/svn/oreilly/hibernate
/current/examples/ch10/classes
codegen:
[hibernatetool]Executing Hibernate Tool with a Standard
Configuration
[hibernatetool]1.task: hbm2java (Generates a set of.java files)
[hibernatetool]16: 46: 38, 403 INFO Environment: 514-Hibernate
3.2.5
[hibernatetool]16: 46: 38, 415 INFO Environment: 547-
hibernate.properties
not found
[hibernatetool]16: 46: 38, 419 INFO Environment: 681-Bytecode
provider name
: cglib
[hibernatetool]16: 46: 38, 434 INFO Environment: 598-using JDK
1.4 java.
sql.Timestamp handling
[hibernatetool]16: 46: 38, 561 INFO Configuration: 1460-
configuring from
file: hibernate.cfg.xml
[hibernatetool]16: 46: 38, 740 INFO Configuration: 553-Reading
mappings
from resource: com/oreilly/hh/data/Track.hbm.xml
[hibernatetool]16: 46: 38, 932 INFO HbmBinder: 300-Mapping class:
com
.oreilly.hh.data.Track->TRACK
.....
[hibernatetool]16: 46: 39, 110 INFO HbmBinder: 1422-Mapping
collection:
com.oreilly.hh.data.Artist.tracks->TRACK_ARTISTS
[hibernatetool]16: 46: 39, 239 INFO Configuration: 1541-
Configured SessionFactor
y: null
[hibernatetool]16: 46: 39, 411 INFO Version: 15-Hibernate Tools
3.2.0.b9
compile:
[javac]Compiling 9 source files
to/Users/jim/svn/oreilly/hibernate
/current/examples/ch10/classes
schema:
[hibernatetool]Executing Hibernate Tool with a Standard
Configuration
[hibernatetool]1.task: hbm2ddl (Generates database schema)
[hibernatetool]16: 46: 41, 502 INFO Configuration: 1460-
configuring from

```

```

file: hibernate.cfg.xml
[hibernatetool]16: 46: 41, 515 INFO Configuration: 553-Reading
mappings
  from resource: com/oreilly/hh/data/Track.hbm.xml
.....
[hibernatetool]16: 46: 41, 629 INFO Configuration: 1541-
Configured SessionFactor
  y: null
[hibernatetool]16: 46: 41, 659 INFO Dialect: 152-Using dialect:
org.hibernate
  .dialect.MySQL5Dialect
[hibernatetool]16: 46: 41, 714 INFO SchemaExport: 154-Running
hbm2ddl
  schema export
[hibernatetool]16: 46: 41, 716 INFO SchemaExport: 179-exporting
generated
  schema to database
[hibernatetool]16: 46: 41, 725 INFO
DriverManagerConnectionProvider: 41-Using
  Hibernate built-in connection pool (not for production use! )
[hibernatetool]16: 46: 41, 726 INFO
DriverManagerConnectionProvider: 42-Hibernat
  e connection pool size: 1
[hibernatetool]16: 46: 41, 727 INFO
DriverManagerConnectionProvider: 45-autocomm
  it mode: false
[hibernatetool]16: 46: 41, 745 INFO
DriverManagerConnectionProvider: 80-using
  driver: com.mysql.jdbc.Driver at URL: jdbc:
mysql: //localhost/notebook_db
[hibernatetool]16: 46: 41, 746 INFO
DriverManagerConnectionProvider: 86-
  connection properties: {user=jim, password=****, shutdown=true}
[hibernatetool]alter table ALBUM_ARTISTS drop foreign key
FK7BA403FC76BBFFF9;
.....
[hibernatetool]alter table TRACK_COMMENTS drop foreign key
FK105B2688E424525B;
[hibernatetool]drop table if exists ALBUM;
[hibernatetool]drop table if exists ALBUM_ARTISTS;
[hibernatetool]drop table if exists ALBUM_COMMENTS;
[hibernatetool]drop table if exists ALBUM_TRACKS;
[hibernatetool]drop table if exists ARTIST;
[hibernatetool]drop table if exists TRACK;
[hibernatetool]drop table if exists TRACK_ARTISTS;
[hibernatetool]drop table if exists TRACK_COMMENTS;
[hibernatetool]create table ALBUM (ALBUM_ID integer not null
auto_increment, TI

```

```

    TLE varchar (255) not null, numDiscs integer, added date, primary
key (ALBUM_ID) )
;
[hibernatetool]create table ALBUM_ARTISTS (ALBUM_ID integer not
null, ARTIST_ID
integer not null, primary key (ALBUM_ID, ARTIST_ID) ) ;
[hibernatetool]create table ALBUM_COMMENTS (ALBUM_ID integer not
null, COMMENT
varchar (255) ) ;
[hibernatetool]create table ALBUM_TRACKS (ALBUM_ID integer not
null, TRACK_ID
integer, disc integer, positionOnDisc integer, LIST_POS integer
not null, primary
key (ALBUM_ID, LIST_POS) ) ;
[hibernatetool]create table ARTIST (ARTIST_ID integer not null
auto_increment,
NAME varchar (255) not null unique, actualArtist integer, primary
key (ARTIST_ID)
) ;
[hibernatetool]create table TRACK (TRACK_ID integer not null
auto_increment, TI
TLE varchar (255) not null, filePath varchar (255) not null,
playTime time, added
date, VOL_LEFT smallint, VOL_RIGHT smallint, sourceMedia varchar
(255) ,
primary key (TRACK_ID) ) ;
[hibernatetool]create table TRACK_ARTISTS (TRACK_ID integer not
null, ARTIST_ID
integer not null, primary key (TRACK_ID, ARTIST_ID) ) ;
[hibernatetool]create table TRACK_COMMENTS (TRACK_ID integer not
null, COMMENT
varchar (255) ) ;
[hibernatetool]create index ALBUM_TITLE on ALBUM (TITLE) ;
[hibernatetool]alter table ALBUM_ARTISTS add index
FK7BA403FC76BBFFF9 (ARTIST_ID) ,
add constraint FK7BA403FC76BBFFF9 foreign key (ARTIST_ID)
references ARTIST
(ARTIST_ID) ;
[hibernatetool]alter table ALBUM_ARTISTS add index
FK7BA403FCF2AD8FDB (ALBUM_ID) ,
add constraint FK7BA403FCF2AD8FDB foreign key (ALBUM_ID)
references ALBUM
(ALBUM_ID) ;
[hibernatetool]alter table ALBUM_COMMENTS add index
FK1E2C21E4F2AD8FDB
(ALBUM_ID) , add constraint FK1E2C21E4F2AD8FDB foreign key
(ALBUM_ID) references
ALBUM (ALBUM_ID) ;

```

```

[hibernatetool]alter table ALBUM_TRACKS add index
FKD1CBBC78E424525B
  (TRACK_ID) , add constraint FKD1CBBC78E424525B foreign key
  (TRACK_ID) references
  TRACK (TRACK_ID) ;
[hibernatetool]alter table ALBUM_TRACKS add index
FKD1CBBC78F2AD8FDB
  (ALBUM_ID) , add constraint FKD1CBBC78F2AD8FDB foreign key
  (ALBUM_ID) references
  ALBUM (ALBUM_ID) ;
[hibernatetool]create index ARTIST_NAME on ARTIST (NAME) ;
[hibernatetool]alter table ARTIST add index FK7395D347A1422D3B
(actualArtist) ,
  add constraint FK7395D347A1422D3B foreign key (actualArtist)
references
  ARTIST (ARTIST_ID) ;
[hibernatetool]create index TRACK_TITLE on TRACK (TITLE) ;
[hibernatetool]alter table TRACK_ARTISTS add index
FK72EFDAD8E424525B
  (TRACK_ID) , add constraint FK72EFDAD8E424525B foreign key
  (TRACK_ID) references
  TRACK (TRACK_ID) ;
[hibernatetool]alter table TRACK_ARTISTS add index
FK72EFDAD876BBFFF9
  (ARTIST_ID) , add constraint FK72EFDAD876BBFFF9 foreign key
  (ARTIST_ID)
  references ARTIST (ARTIST_ID) ;
[hibernatetool]alter table TRACK_COMMENTS add index
FK105B2688E424525B
  (TRACK_ID) , add constraint FK105B2688E424525B foreign key
  (TRACK_ID) references
  TRACK (TRACK_ID) ;
[hibernatetool]16: 46: 42, 630 INFO SchemaExport: 196-schema
export complete
[hibernatetool]16: 46: 42, 631 INFO
DriverManagerConnectionProvider: 147-cleanin
g up connection pool: jdbc: mysql: //localhost/notebook_db
[hibernatetool]9 errors occurred while performing<hbm2ddl>.
[hibernatetool]Error#1:
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: T
able'notebook_db.album_artists'doesn't exist
[hibernatetool]Error#1:
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: T
able'notebook_db.album_artists'doesn't exist
[hibernatetool]Error#1:
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: T
able'notebook_db.album_comments'doesn't exist

```



```
[hibernatetool]Error#1:
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: T
able'notebook_db.album_tracks'doesn't exist
[hibernatetool]Error#1:
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: T
able'notebook_db.album_tracks'doesn't exist
[hibernatetool]Error#1:
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: T
able'notebook_db.artist'doesn't exist
[hibernatetool]Error#1:
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: T
able'notebook_db.track_artists'doesn't exist
[hibernatetool]Error#1:
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: T
able'notebook_db.track_artists'doesn't exist
[hibernatetool]Error#1:
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: T
able'notebook_db.track_comments'doesn't exist
BUILD SUCCESSFUL
Total time: 8 seconds
```

发生了什么事

Hibernate会自行配置和使用**MySQL**的特定功能，检查我们的**Track**类的映射文档，连接到**MySQL**服务器，执行必要的命令以创建供持久化我们的示例数据所需要的数据库模式（和以前一样，最后会报告几个**SQL**异常，你可以忽略它们；这些异常可能是因为**Hibernate**试图删除并不存在的外键而引起的，因为这是我们首次尝试创建数据库模式）。

再次说明一下，不必担心最后出现的错误；这些错误是由于为了防止已经存在部分数据库模式，**Hibernate**抢先试图删除它们而造成

的。即便报告有错误，**Hibernate**并不认为它们会造成问题，所以继续处理剩下的数据库模式创建任务，并最终报告操作成功完成。

把这个示例和**HSQLDB**版本的数据库模式创建（例2-7）进行比较，结果会很有趣。它们的输出差不多是相同的，但是用于创建数据库表的**SQL**语句明显不同。这就是**Hibernate**中所谓的**SQL“方言”**。

查询数据

有了数据库服务器，你就可以再次运行MySQL客户端，确认已经创建好了Track类映射的数据库模式，结果如例10-5所示。

例10-5：检查新创建的MySQL数据库模式

```
% mysql -u jim -p
```

```
Enter password:
```

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
```

```
Your MySQL connection id is 21 to server version: 5.0.27
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

```
mysql> use notebook_db
```

```
Reading table information for completion of table and column names
```

```
You can turn off this feature to get a quicker startup with -A
```

```
Database changed
```

```
mysql> show tables;
```

```
+-----+
| Tables_in_notebook_db |
+-----+
| ALBUM_ARTISTS          |
| ALBUM_COMMENTS         |
| ALBUM_TRACKS           |
| ARTIST                 |
| TRACK_ARTISTS          |
| TRACK_COMMENTS         |
| album                  |
| track                  |
+-----+
8 rows in set (0.00 sec)
```

```
mysql> describe track;
```

```
+-----+-----+-----+-----+-----+-----+
| Field      | Type      | Null | Key | Default | Extra      |
+-----+-----+-----+-----+-----+-----+
| TRACK_ID   | int(11)   | NO   | PRI | NULL    | auto_increment |
| TITLE      | varchar(255) | NO   | MUL |         |              |
| filePath   | varchar(255) | NO   |     |         |              |
| playTime   | time      | YES  |     | NULL    |              |
+-----+-----+-----+-----+-----+-----+
```

```

| added      | date      | YES |      | NULL |      |
| VOL_LEFT   | smallint(6) | YES |      | NULL |      |
| VOL_RIGHT  | smallint(6) | YES |      | NULL |      |
| sourceMedia | varchar(255) | YES |      | NULL |      |
+-----+-----+-----+-----+-----+-----+
8 rows in set (0.03 sec)

mysql> select * from TRACK;
Empty set (0.00 sec)

mysql> describe album_tracks;
+-----+-----+-----+-----+-----+-----+
| Field          | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| ALBUM_ID       | int(11)   | NO   | PRI |          |       |
| TRACK_ID       | int(11)   | YES  | MUL | NULL     |       |
| disc           | int(11)   | YES  |     | NULL     |       |
| positionOnDisc | int(11)   | YES  |     | NULL     |       |
| LIST_POS       | int(11)   | NO   | PRI |          |       |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> quit;
Bye

```

发现数据表是空的，这并不奇怪。如果你从示例所在的目录中运行`ant ctest`命令，接着再尝试做个`select`查询，将会看到类似例10-6所示的数据。只需要修改一个Hibernate使用的配置文件，就可以完全切换所使用的数据库。当项目进行中客户多次提出要改变他们需要的数据来源时，或是多位开发人员各自喜欢使用不同的操作系统时，以这种方式来修改配置显得非常方便。

例10-6： 在运行`ctest`构建目标后，查询TRACK数据库表

```
mysql> select * from track;
```

TRACK_ID	TITLE	playTime	added	VOL_LEFT	VOL_RIGHT	sourceMedia	filePath
1	Russian Trance	00:03:30	2007-09-22	100	100	CD	vol2/album610/track02.mp3
2	Video Killed the Radio Star	00:03:49	2007-09-22	100	100	VHS	vol2/album611/track12.mp3
3	Gravity's Angel	00:06:06	2007-09-22	100	100	CD	vol2/album175/track03.mp3
4	Adagio for Strings (Ferry Corsten Remix)	00:06:35	2007-09-22	100	100	CD	vol2/album972/track01.mp3
5	Adagio for Strings (ATB Remix)	00:07:39	2007-09-22	100	100	CD	vol2/album972/track02.mp3
6	The World '99	00:07:05	2007-09-22	100	100	STREAM	vol2/singles/pvw99.mp3
7	Test Tone 1	00:00:10	2007-09-22	50	75	NULL	vol2/singles/test01.mp3

```
7 rows in set (0.00 sec)
```

其他

在图形界面中使用MySQL？如果你喜欢使用像我们在演示HSQLDB时用的那种图形界面工具，可以试试MySQL GUI工具，可以从<http://dev.mysql.com/downloads/gui-tools/5.0.html>下载它。

连接到Oracle数据库，或其他你喜欢的数据库，或是共享的、遗留的数据库，总之不是MySQL或HSQLDB数据库？你可能已经想到连接到其他数据库也同样很简单，所有需要做的就是修改hibernate.cfg.xml中的hibernate.dialect属性，以反映你想使用的那种数据库。可供选用的

数据库方言有许多种，差不多已经包括了我能够想得到的每种免费的和商业的数据库。附录C列出了在编写本书时Hibernate能够支持的所有数据库方言，不过还是应该查阅Hibernate的官方文档以得到最新的列表。如果你想使用比较冷僻的数据库，那可能就得自己编写支持它的数据库方言了，但是这种情况看起来不太可能发生（最好先查查是否已经有人开始为之付出努力了）。

在选择好使用哪种数据库方言以后，还需要正确设置hibernate.connection的各个属性（driver、URL、username以及password），才可以建立连接到你所选择的数据库环境的JDBC连接。如果你是在移植现有的项目以使用Hibernate，同可以从项目的代码或配置中得到这些信息。而且自然地，在项目构建和运行期间，必须提供项目依赖的JDBC驱动程序。

当然，如果你正在连接一个现有的或共享的数据库，就不必使用Hibernate来创建数据库模式了。相反，这时你需要编写Hibernate映射文档，以便将现有的数据库模式映射到持久化类。这一步可以手工完成，或者使用Hibernate Tools（我们将在第11章深入研究这一工具）的帮助，还可以使用像Middlegen（[\[1\]](#)）之类的第三方工具包，之后就可以按持久化对象的形式来使用数据了。

使用Hibernate甚至还可以同时与多种不同的数据库进行交互，只需要用单独的配置文件来创建多个会话工厂实例。这样的使用方法已

经超出了本书所要演示的简单、自动化的配置范围，不过，Hibernate 参考文档中有这方面的例子。当然，在某一时刻，一个持久化对象只能与一个会话实例相关联，这意味着一个持久化对象一次只能链接到一个数据库。不过，即使不同的数据库使用不同的模式来表示持久化类，还是可以通过精心设计的编码，实现在不同的数据库系统之间复制或移动对象。当然，这样复杂的处理更是超出本书的讨论范围了！

[1] <http://boss.bekk.no/boss/middlegen/>.

第11章 Hibernate与Eclipse: Hibernate Tools使用实战

在Eclipse中安装Hibernate Tools

如果你是使用Eclipse作为Java开发环境（现在很多开发人员都在使用它（[\[1\]](#)）），和本书其他部分对这一工具的使用（我们原来用它为Ant构建处理增加Hibernate相关的功能）相比，在Eclipse中可以更加充分地使用Hibernate Tools。

应该怎么做

在深入讨论以前，先需要保证你使用的Eclipse的版本足够新。目前Hibernate Tools的发行版本至少需要Eclipse 3.3和WTP 2.0。所以，如果你的版本不够新，没有理由不借这个机会更新一下。

在Eclipse中安装Hibernate Tools最简单的方法就是通过Eclipse普通的网站更新机制。首先需要告诉Eclipse到哪找Hibernate Tools，在"Software Update"菜单中选择"Find and Install"菜单项，如图11-1所示。

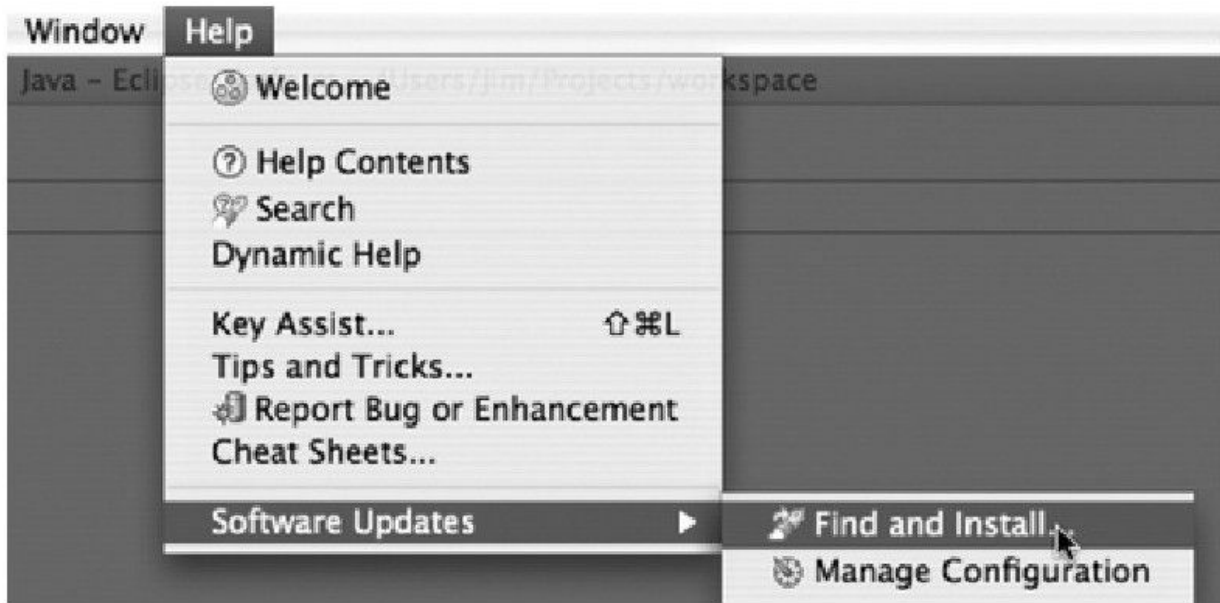


图 11-1 Eclipse的更新功能

我们现在需要安装全新的一个插件，而不是更新现有的插件，所以在接下来的窗口中应该选择"Search for new features to install"，再点击"Next"。

Eclipse自己不会自动知道Hibernate Tools更新网站，所以我们需要告诉它到哪去找。点击"New Remote Site"，如图11-2所示。



图 11-2 在Eclipse中从一个新的更新网站来安装插件

输入一些描述性的文字（如"Hibernate Tools"），以作为新的更新网站的名称。Eclipse现在需要我们指定更新网站的URL，如稍后的图11-4所示。我们必需在网上搜索一下，才能找到正确的更新URL。

切换到Web浏览器，打开Hibernate网站<http://hibernate.org>，在页面左边的导航菜单中点击"Hibernate Tools"（或者，你也可以直接打开<http://tools.hibernate.org>这个目录链接）。在打开页面以后，可以看看有

关这一工具的介绍，以及各种文档链接。我们在第一次研究怎么把这个工具安装到Eclipse中时，还花了些时间，最后才发现我们错过了一直要找的东西：在Tools页面上原本就有一个更新网站链接。将这个链接的URL复制到剪切板，如图11-3所示（在链接上用鼠标右键点击或按住Ctrl键点击，就可以打开上下文关联菜单）。

Hibernate Tools for Eclipse and Ant

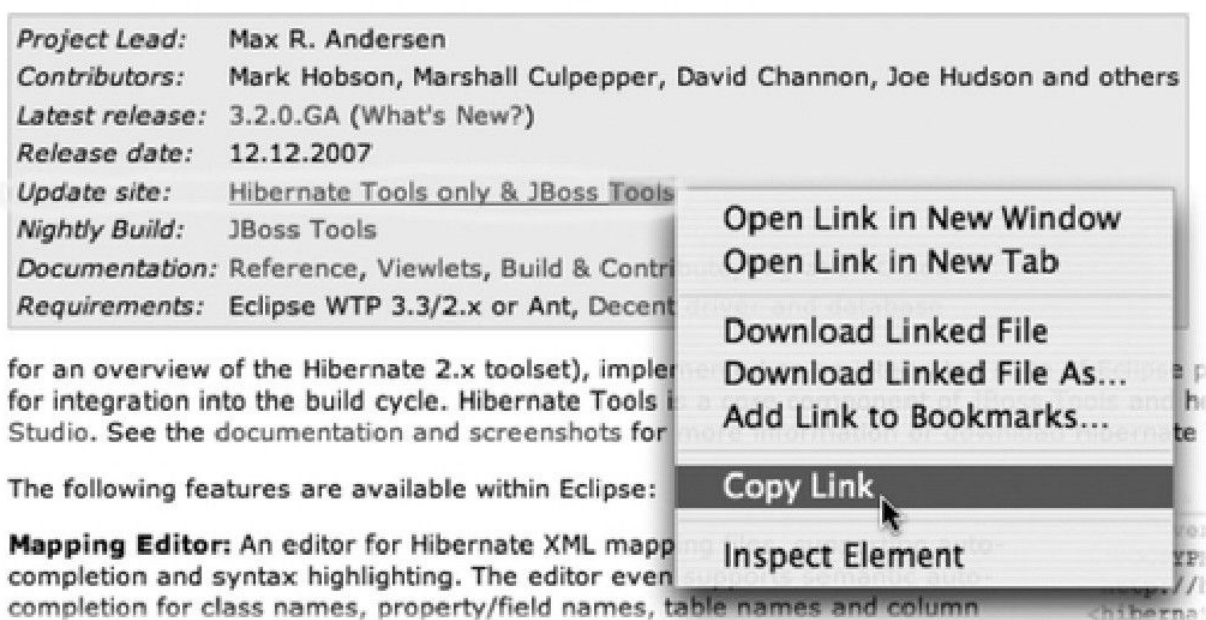


图 11-3 获取Hibernate Tools更新的网站URL

如果你仔细观察，可能会注意到这个页面上Hibernate Tools的版本要比我们这本书使用的版本（3.2.0 Beta9a）还新。不错，你现在应该明白像Hibernate这样大型开源项目变化得太快了，写本关于它的书有多么难！在编写本书时，Hibernate Tools的稳定版本才刚推出，Maven库中还未提供对这个版本的支持，所以我们现在先不用管本书原来使

用的Hibernate Tools。无论如何，新版本的变化都不会影响其他章节的示例，只是Eclipse的当前最新版本（编写本书时的版本是3.3，也称为Europa）需要配合使用Hibernate Tools的最新版本，这样才可以支持本章将要演示的一些功能。

将更新网站的URL复制粘贴到Eclipse的"New Update Site"窗口中，如图11-4所示。点击"OK"按钮，以确认保存这个新的网站URL。



图 11-4 在Eclipse中配置一个新的更新网站

从图11-5中可以看到，Eclipse合理地假设你想使用刚才配置的更新网站（勾选了Hibernate Tools），所以，这时只要点击"Finish"按钮，Eclipse就会自动连接更新网站，检查从那可以安装的插件。

接着会弹出一个窗口（如图11-6所示），这个窗口比较简单，它的出现就表明胜利在望了。所有我们需要做的就是勾选Hibernate Tools节点旁边的复选框，现在就安装，对吗？



图 11-5 新的更新网站已经准备好使用



图 11-6 正在安装Hibernate Tools：我们需要再接再厉

很不走运，当我们选中Hibernate Tools节点旁边的复选框时，遇到了麻烦。这个窗口顶部显示了一条错误消息，"Next"按钮也变成了灰色，不让我们继续使用了。

出了什么问题

Eclipse不让我们继续安装，是因为SeamTools功能（组成Hibernate Tools工具集的10个插件之一）需要安装其他插件

（org.eclipse.datatools.connectivity.feature），而且这些插件还不能默认安装。非常不幸，我们只得后退几步。点击"Cancel"按钮，再从本节开始介绍的"Find and Install"菜单选项开始。再一次选择"Search for new features to install"，点击"Next"。这次要选中Europa Discovery Site节点，如图11-7所示，再点击"Finish"按钮。



图 11-7 选中Eclipse Discovery Site

从接下来出现的下载镜像列表选择一个与你的位置合适的选项，点击"OK"按钮（如果你配置Eclipse自动选择下载镜像，则不会出

现这一步)。这一次你将看到两组插件: **Europa Discovery Site**和**Hibernate Tools**。再次选择**Hibernate Tools**这组插件来安装。和以前一样,窗口顶部还是会出现错误提示,但是这次我们有机会可以修复这个问题。一种办法就是选择整个"**Europa Discovery Site**"插件组,但是这样的下载量将大大超过我们实际的需要,会让**Eclipse**的配置变得过于庞大。

所以,我们得规划出需要从**Europa Discovery Site**节点下载的最少的一组插件。保持**Hibernate Tools**插件节点为选中状态,展开**Europa Discovery Site**插件组节点,但是不选中任何东西。虽然可以手工查找和选择**Hibernate Tools**报告需要的插件,但这会导致大量繁琐地摸索、尝试、解决错误,才能确切地明白哪些插件的组合才能让我们需要的东西正常工作(我们试验时,曾经找到过第一个错误中提及的提供数据工具连接功能的插件,不过,我们发现这个插件也有它自己要依赖的其他插件,看来这个插件还不是我们必需的惟一插件)。就在我们摩拳擦掌,准备详细写写如何精确选择需要下载的插件时,我们注意到界面右边的"**Select Required**"按钮,其实可以用它来自动选择需要的所有插件。不过,我们的实验表明,只有当你第一次打开相应网站(将从这个网站下载你需要的插件)旁边的“打开”三角形节点时, "**Select Required**"按钮才会起作用。直到我们展开"**Europa Discovery site**"这个节点以前,它什么也不会做。在展开这个节点以后,点击"**Select Required**"按钮会自动选择解决依赖错误所需要的插件。既然

现在你已经选中了Hibernate Tools节点，Europa Discovery site节点也展开了，那就点击一下"Select Required"按钮吧。最后结果应该类似于图11-8所示，不过，要下载安装的插件的确切个数将依赖于原来你可能已经安装过的插件。

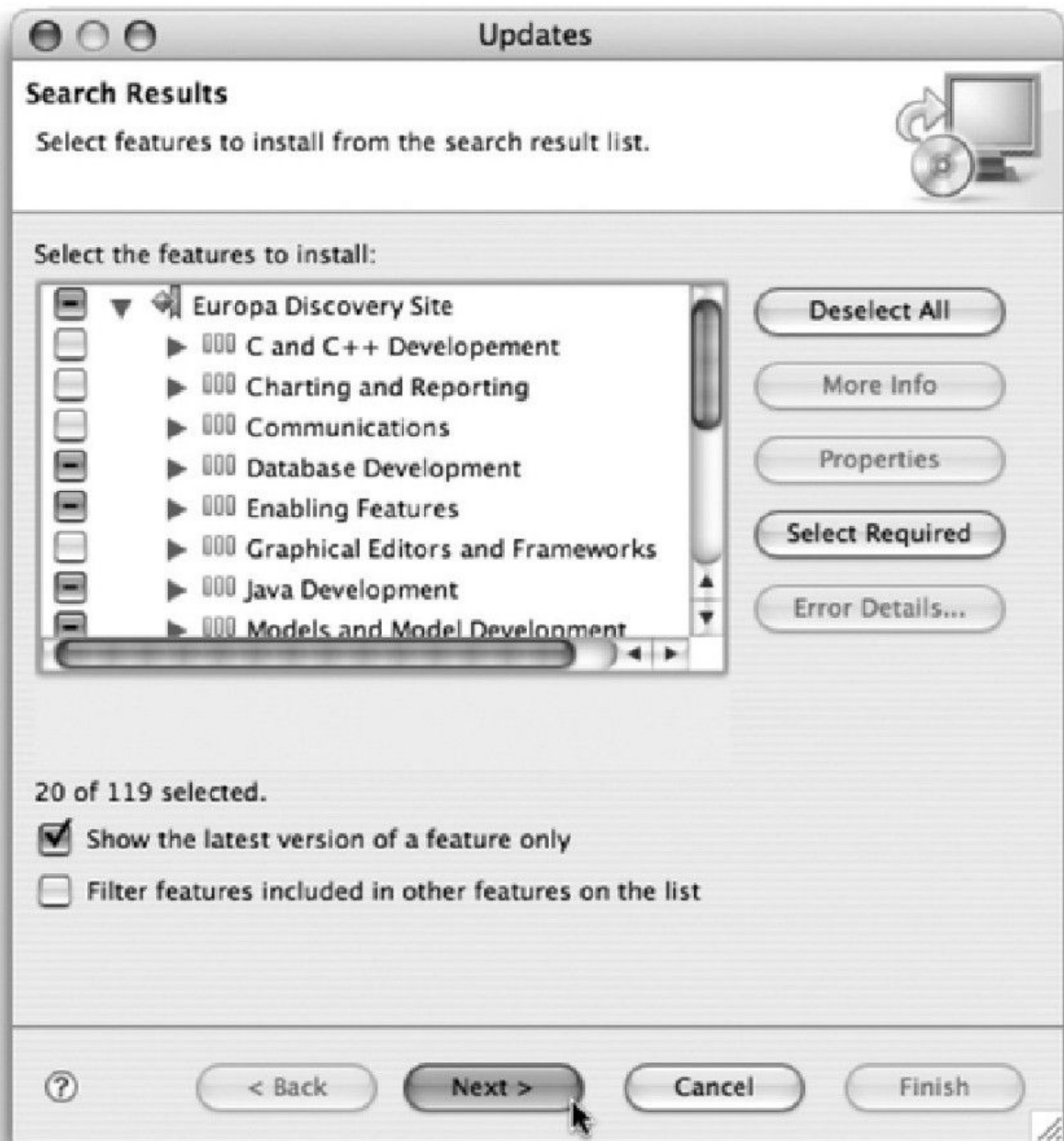


图 11-8 选中需要的所有插件

这次不会再有任何错误消息妨碍我们安装Hibernate Tools了！点击"Next"按钮，就开始安装这组插件，接受在下一个对话框中的许可协议，并再次点击"Next"。之后会再弹出一个对话框，向你显示一组可以选择安装的可选功能。因为我们只需要Hibernate Tools就足够了，所以在Optional Features（可选功能）对话框中直接点击"Next"按钮以便继续安装我们需要的插件。默认的安装位置（具体位置随Eclipse安装位置的不同而不同）差不多总是不错的，除非有特殊原因才需要设置到其他位置，点击Installation（安装）对话框的"Finish"按钮，继续完成插件的安装。

在下载更新期间，Update Manager（更新管理器）将一直持续运行。最后，将弹出一个Feature Verification（功能确认）对话框。对于正在安装的每个功能，这个对话框将报告每个功能是否具有Eclipse项目授权的机构颁发的加密签名。一些插件具有这样的签名，而另一些（像Hibernate Tools本身）则没有有效的签名。第三方Eclipse插件通常都是这样的。Eclipse只是对这种情况进行警告，希望确认你愿意安装其他网站的插件。因为我信任提供要下载的插件的网站，所以全部都确认继续安装。如果你也像我这样（基本上，如果你想使用Hibernate Tools，就得信任提供这些插件的网站），就只需要点击"Install"按钮，以继续安装某个插件（如果你不想对每个插件进行确认，可以点

击"Install All"按钮，以批量接受所有插件）。Update Manager（更新管理器）会继续安装。在安装完所有插件后，它会建议你重新启动Eclipse。为了确保安全，点击"Yes"按钮，等待Eclipse重新启动。

现在做什么

很明显，与过去相比，Eclipse现在对Hibernate提供了更多的支持。Hibernate配置文件的图标现在包含了一个小的Hibernate标志，双击这个图标或映射文件，会打开一个特殊的编辑器，在这个编辑器中创建和更新Hibernate映射文件会更方便，而不用直接处理底层的XML（图11-9）。

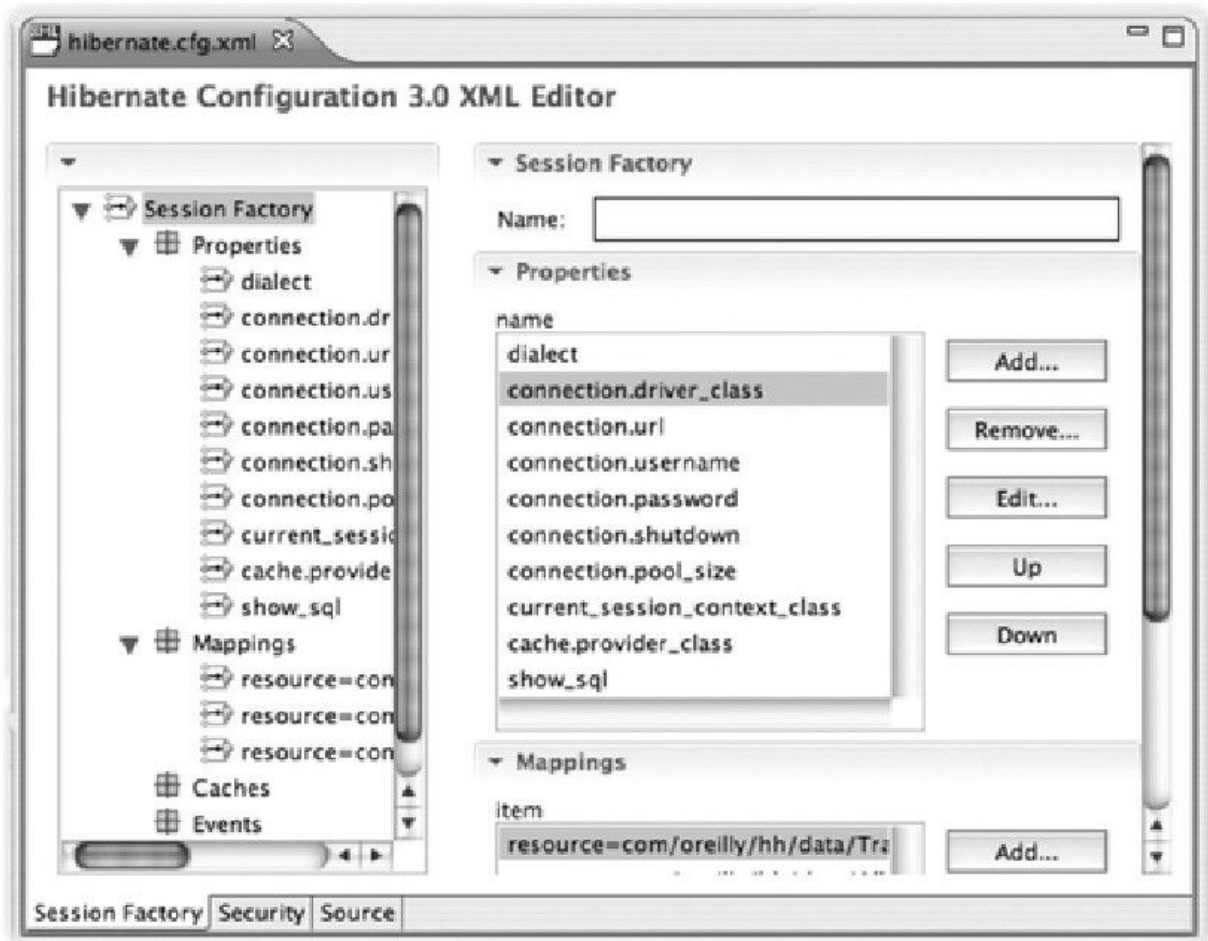


图 11-9 Hibernate配置文件编辑器

注意：这样就不用像在源代码视图下，得频繁查阅参考文档才可以编辑了！

当然，如果你觉得直接处理XML更高效，可以点击编辑器底部的"Source"（源代码）选项页，仍然可以查看XML。在源代码视图中编辑时，你会发现它已经为编辑各种Hibernate相关元素以及取值提供了辅助完成（completion assistance）的功能，如图11-10所示。这并不是XML编辑器普通的元素名称自动完成功能，普通自动完成只是通过分

析XML DTD实现的，而DTD缺乏对属性名称的足够描述，只能规定些文本规则。

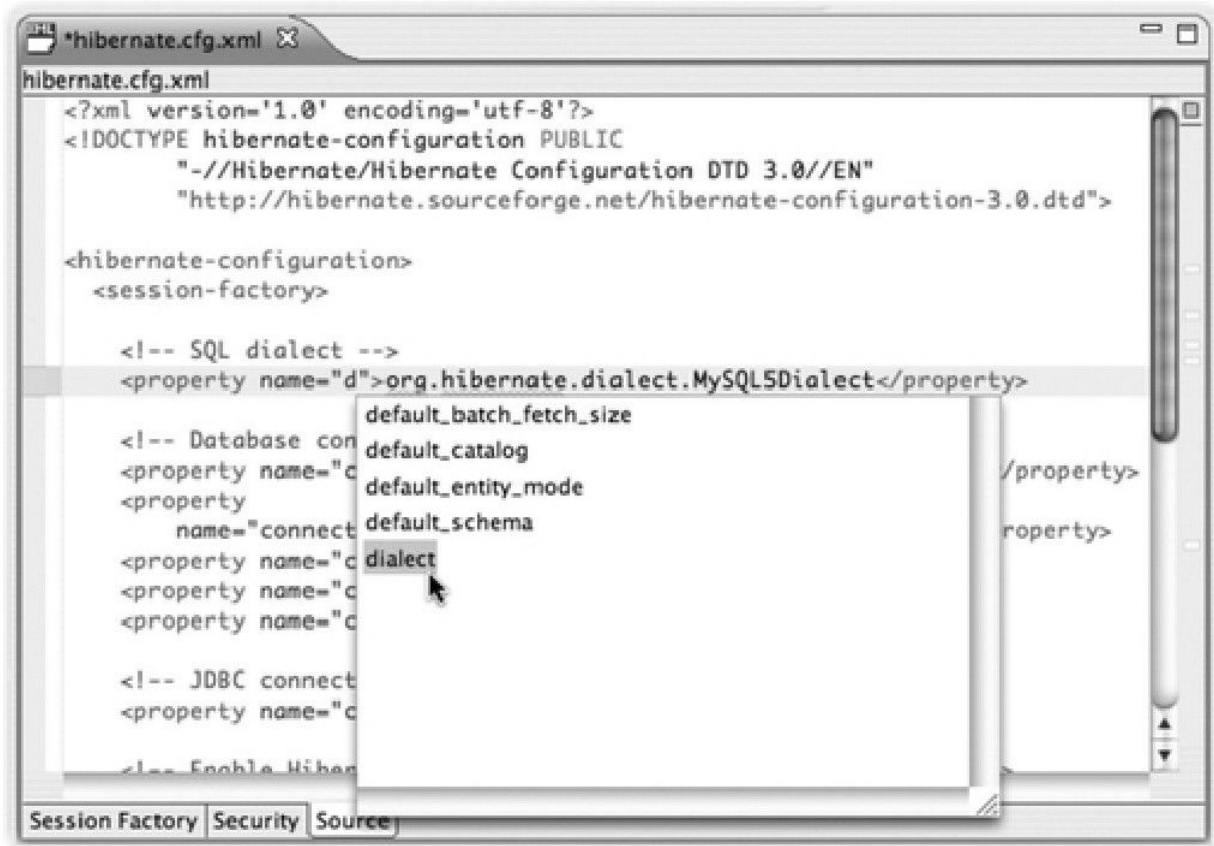


图 11-10 Hibernate配置文件编辑器中的属性名称自动完成功能

你可以使用普通的Eclipse自动完成组合键（ControlSpace）来打开相应的弹出窗口。

映射文档编辑器如图11-11所示。这两个编辑器看起来功能强大也很有用；值得花些时间来搞明白它们的工作原理和功能。对于在Eclipse中使用Hibernate来说，它们本身也是很有价值的工具。

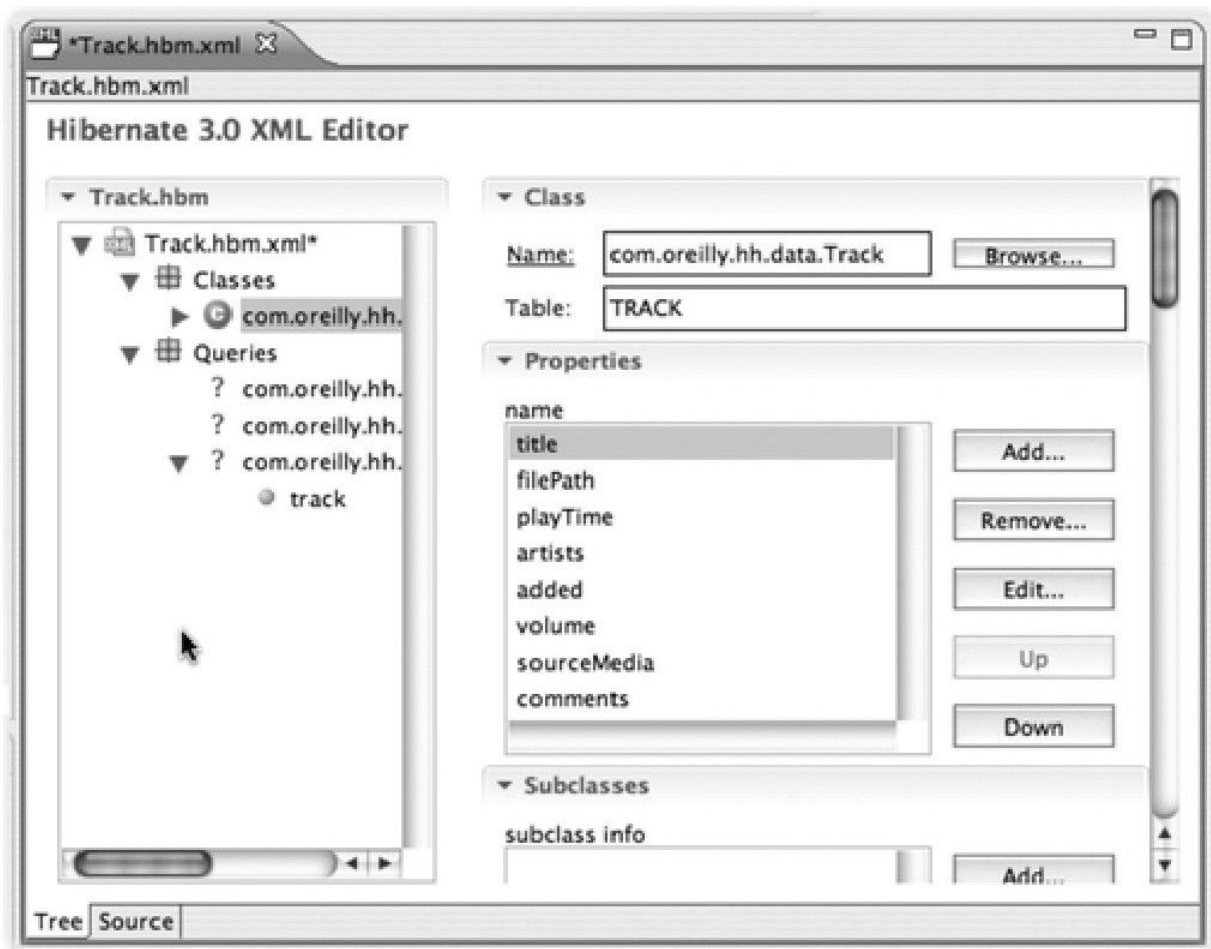


图 11-11 Hibernate映射编辑器

为了探索用Hibernate Tools还可以做些什么其他事，首先要在项目中激活它。如果你是从一个新项目开始的，可以选择先创建一个新的Hibernate配置文件。如果你是从一个现有的Hibernate项目开始的（本章我们就是这样做的），你可以跳过这一步，只需要创建一个新的Hibernate控制台（console）就可以了。

[1] http://www.oreillynet.com/onjava/blog/2004/06/ive_been_eclipsed.html.

创建一个Hibernate控制台配置

在项目浏览器中选择项目，选择**File → New → Other**，扩展由工具添加的**Hibernate**节点（如图11-12），选择**Hibernate Console Configuration**（从这里能够选择创建一个新的配置文件、XML映射文件以及对配置文件进行逆向工程处理（本书不介绍这个主题））。点击**"Next"**以继续为项目安装这一工具。

注意：如果你仍然还在使用旧的**properties**属性文件来配置**Hibernate**，则可以使用这个例子上面的**"Property file"**。



图 11-12 Hibernate Tools提供的新的Eclipse向导

Hibernate Console Configuration窗口打开以后（如图11-13所示），先要通过点击相应的“Browse”（浏览）按钮，再在项目中选定一个文件，告诉工具到哪儿可以找到你的Hibernate配置文件（当第一次使用配置文件向导来创建配置文件时，会自动创建好配置文件，最新版本

的Hibernate Tools工具看起来足够智能，可以自己在本书代码示例的目录结构中找到配置文件）。

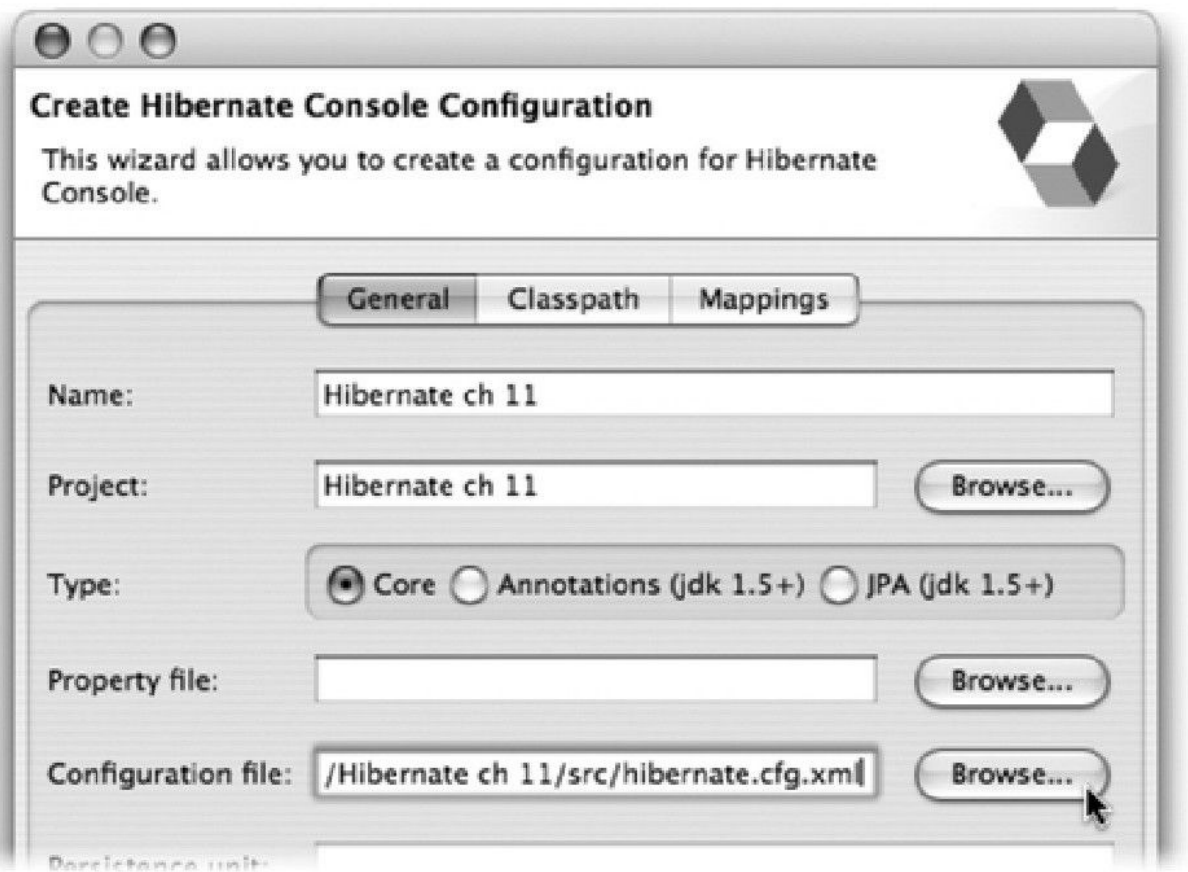


图 11-13 建立Hibernate Console配置

对于大部分项目来说，这个选项卡（tab）中的其他设置可以不用修改，不过，因为我们使用Maven Tasks for Ant来管理我们的依赖库，而Hibernate不会奇迹般地知道到哪可以找到Maven仓库中的数据库驱动程序，所以需要调整类路径（Classpath）的设置。点击"Classpath"选项卡，在这个选项卡上再点击"Add External JARS"按钮，手工告诉工具到哪找数据库驱动程序，如图11-14所示。

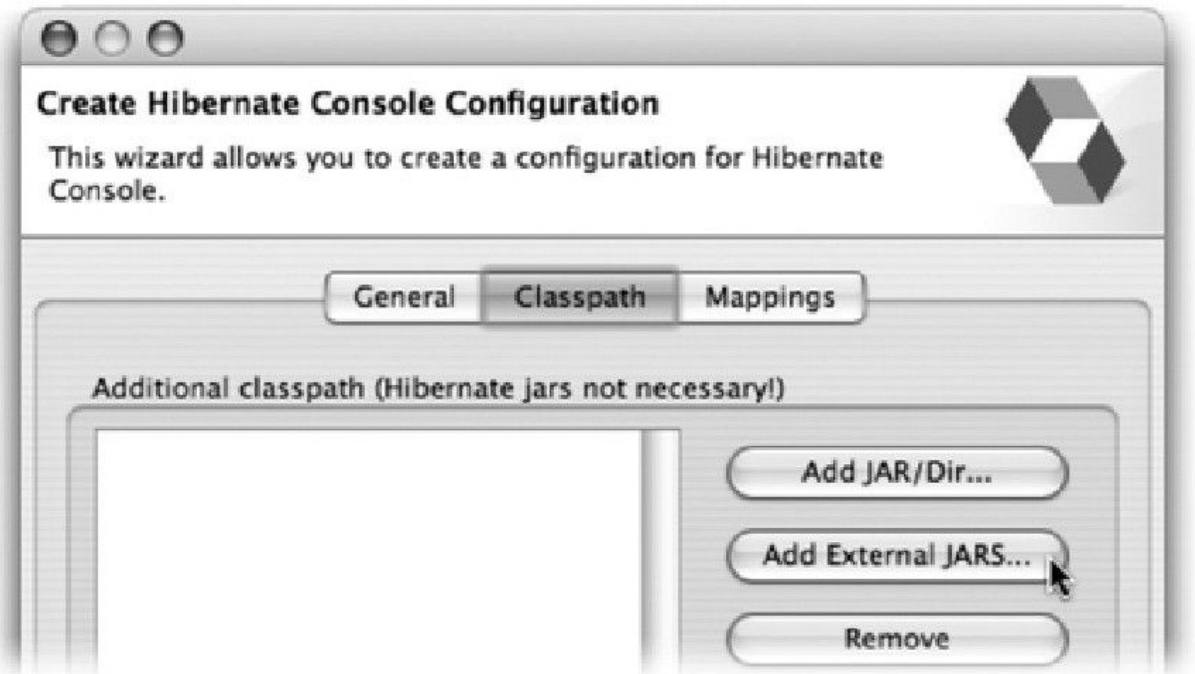


图 11-14 Hibernate Console类路径配置

注意：可以按共享模式（shared mode）来使用HSQLDB，提供多个数据库连接，不过这部分不在本书讨论范围内，而且你还需要考虑在哪以及如何运行“服务器”JVM。

在打开的文件选择对话框中，导航到Maven仓库的目录（可以参阅第1章1.7节结尾部分的介绍），找到MySQL驱动程序。在这个例子中，如果是在Mac OS X操作系统下，驱动程序应该位于
~/.m2/repository/mysql/mysql-connector-java/5.0.5/mysql-connector-java-5.0.5.jar。在上一章的基础上，我们继续使用MySQL。用Hibernate Tools连接外部数据库要方便得多，这时再像我们原来那样，试图使用嵌入在内存中的数据库，就不是个好想法了，因为Hibernate Tools会认

为它们可以自由地将到数据库的连接保持为打开状态，这样，当你想对数据库进行其他操作，比如通过Ant来运行db构建任务来查看数据库的内容时，就不得不先退出Eclipse。相反，像MySQL这样独立的数据库，在处理多个同时并发的数据库连接时，就没有这样的问题。

选择好驱动程序的JAR文件以后，点击"Open"按钮（如图11-15所示）。

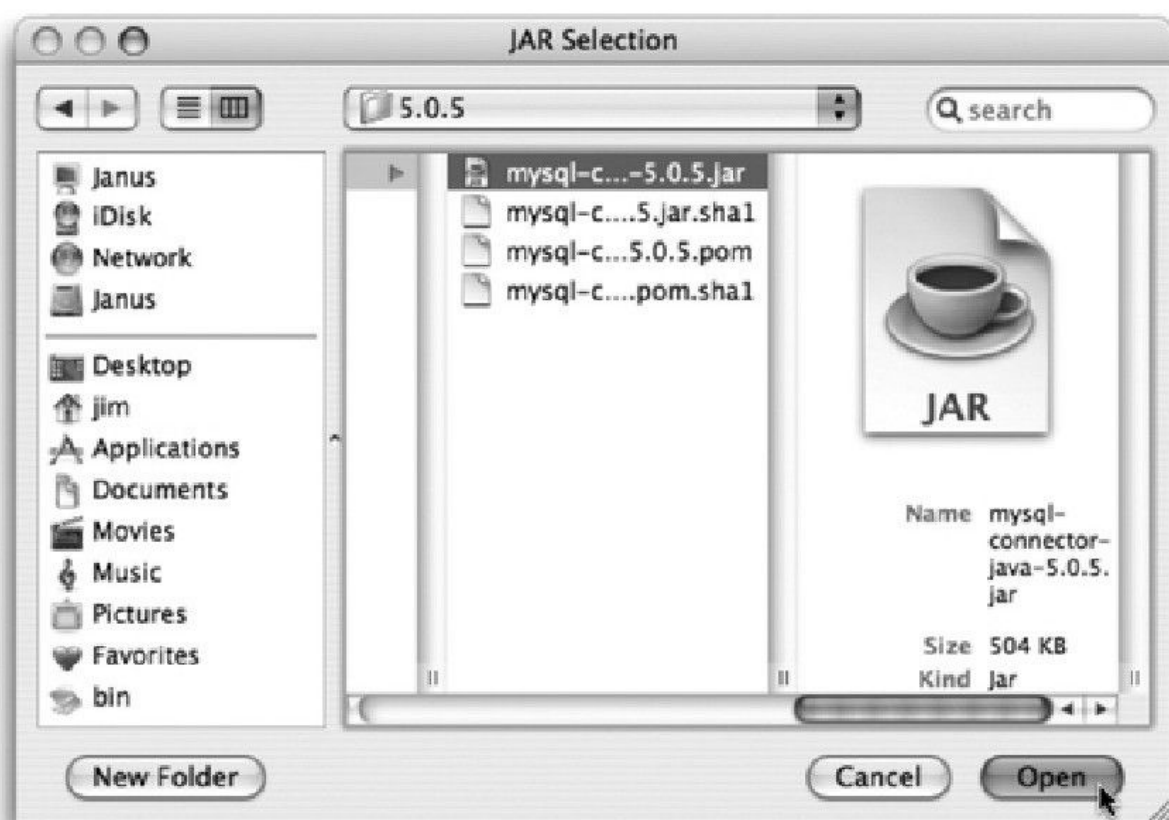


图 11-15 为Hibernate Console配置加载数据库驱动程序JAR文件

注意：如果你在使用像Oracle这样版权专有的数据库，可能就需要自己手工下载JDBC驱动程序了。Maven仓库只提供自由软件。

在配置好类路径以后（如图11-16所示），就可以点击"Finish"按钮，准备创建我们的Hibernate Console配置。



图 11-16 Hibernate Console类路径配置完毕

Hibernate Tools会询问我们是否想为项目添加对Hibernate的支持（如图11-17所示）。好哇，那就是我们盼望的啊！马上点击"OK"按钮。

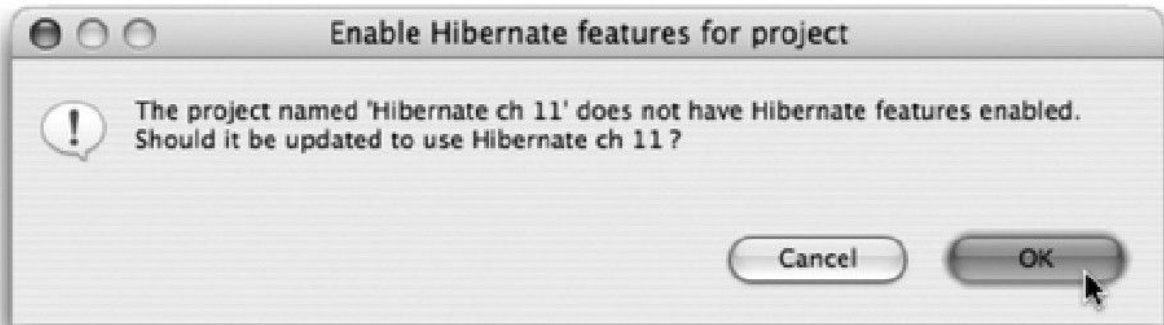


图 11-17 Hibernate功能准备好了

更多的编辑支持

我们的项目现在已经支持**Hibernate Tools**了，那可以用它做什么呢？嗯，现在，当你编辑映射文件时，**XML**编辑器就能够帮你自动完成数据表和列的名称（如图**11-18**所示）。这也正是为什么需要将项目关联到**Hibernate Console**配置的原因：**Hibernate Tools**实际上维护着一个**Hibernate**会话，通过它来检查数据库模式，为实际项目环境提供相关的帮助。

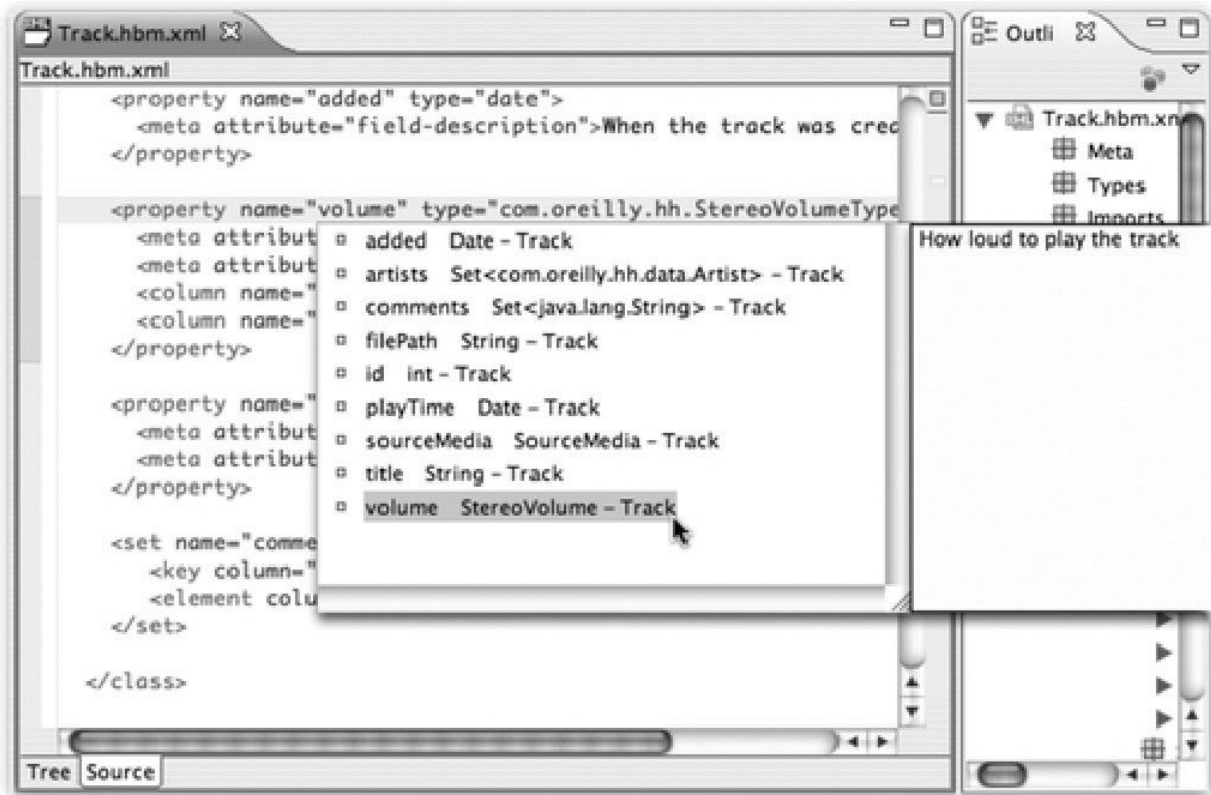


图 11-19 Hibernate映射编辑器中属性名称的自动完成

对于数据库驱动自动完成，我们确实遇到了一个“意想不到的麻烦”（gotcha）。即便SQL通常是不区分数据表和字段名称的大小写，而Hibernate需要区分。我们的曲目数据表全是用小写字母创建的，而映射文档都是用大写字母引用的，如果不注意到这一点，就不能正常使用自动完成。修改映射文档，让它与数据库模式定义中真实的大小写情况相匹配，就可以解决问题了。

这里还可以使用Eclipse的另一个有用功能，F3快捷键，它用于导航跳转到变量、方法以及类的声明位置，在映射编辑器中使用这个快捷键可以把你带到正在映射到的类或属性的Java代码定义中。

我们惊喜地发现，编辑器还支持自定义的类型映射，如图11-20中类型自动完成菜单的底部所示，看起来与GUI映射编辑器有些不同。



图 11-20 自定义类型映射的自动完成

稍等，还有更多

这些只是Eclipse的普通功能，在Java视图下就可以使用。不过，在打开Hibernate Console视图后，还有更多其他窍门可以应用。要打开这个视图，可以点击工具条上的"Open Perspective"按钮，再选择"Other"（其他）选项，或者选择Window → Open Perspective → Other。

这两种方法都会打开"Open Perspective"对话框，如图11-21所示。选择"Hibernate Console"，再点击"OK"按钮。

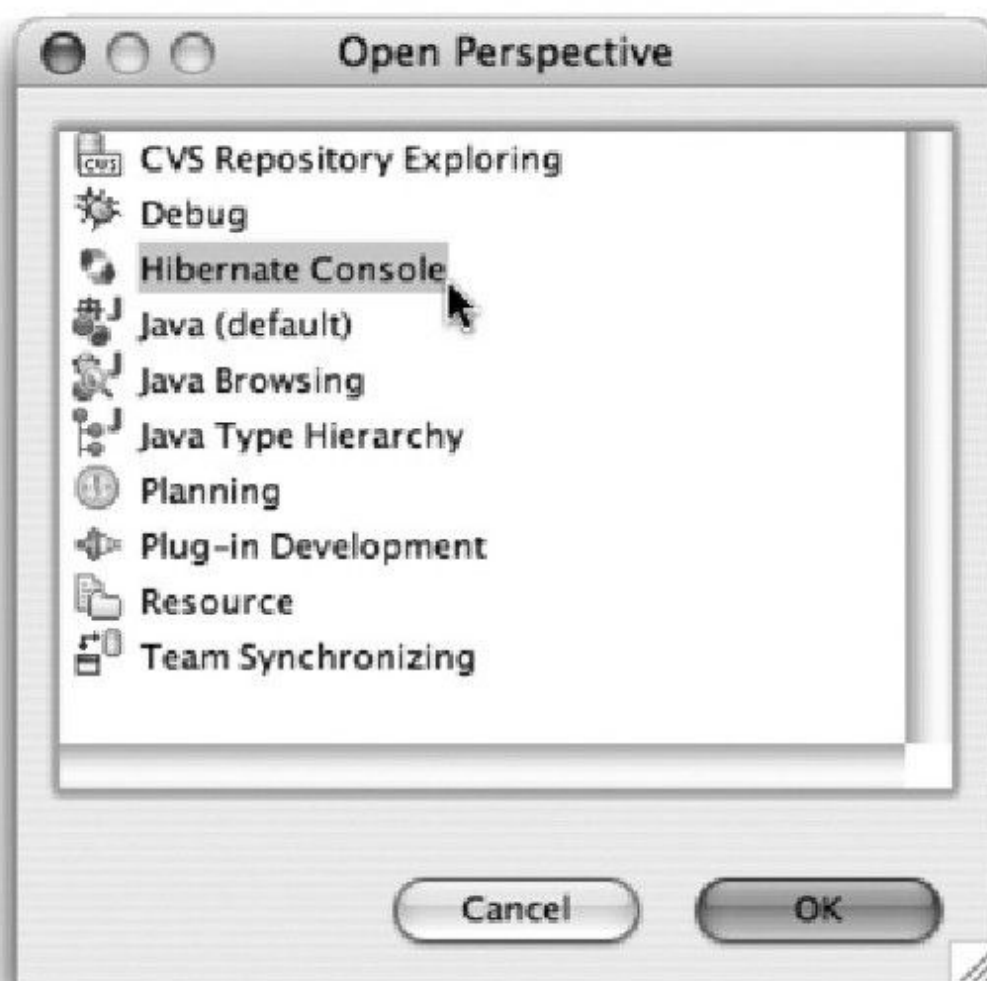


图 11-21 打开Hibernate Console视图

Hibernate Console视图

首先要注意的是图11-22所示的Hibernate Configurations（Hibernate配置）视图。图中，我们已经展开了本节开始创建的"Hibernate ch 11"配置节点，看看怎么使用其中的映射、类、数据库模式等条目。注意图中为主键（**identifier**）、多对一、一对多关联分配的标识图标。在这个视图中提供了丰富的有用信息。打开这个视图的下拉菜单，可以看到通过它可以访问几个有趣的功能（同时也解释了菜单左边各按钮的功能目的）。

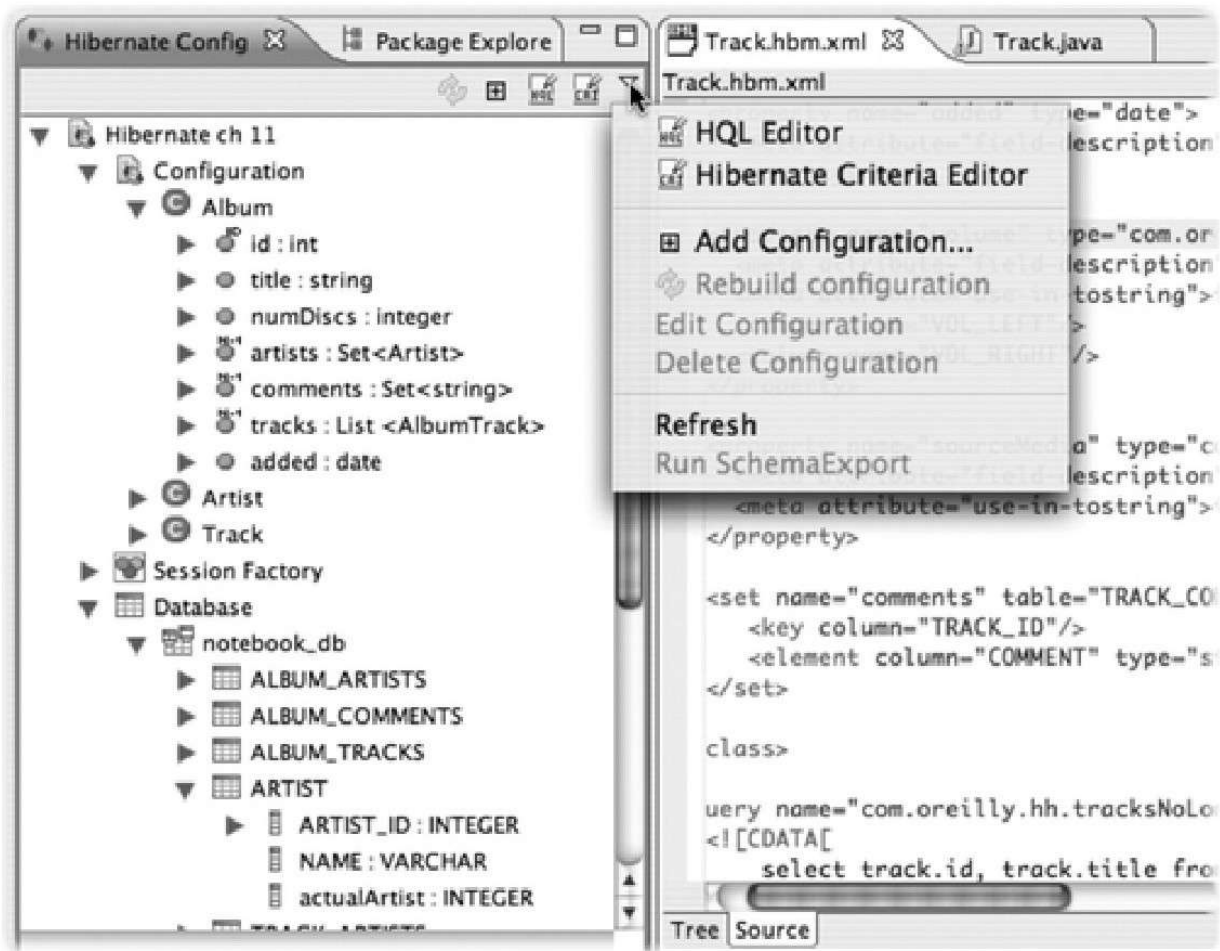


图 11-22 Hibernate Console视图下的Hibernate配置界面

以简单的方式，这一菜单可以让我们编辑现有的Hibernate配置、为其他Hibernate项目创建新的配置，如果在Eclipse环境以外修改了某些文件，还可以刷新这一视图。更为强大的是，"Run Schema Export"选项可以让我们只要简单地选中某个配置项，再运行这个菜单选项，就可以得到与第2章编写的schema Ant构建任务同样的结果。

这一菜单为它左边那些看起来很神秘的按钮的作用提供了些提示说明。接下来我们看看这些按钮可以完成什么复杂功能，首先从"HQL

Editor"开始，图11-23演示了选择菜单项并点击按钮后，将会显示的界面。看起来我们应该能够在这个地方处理HQL，但是使用Control Space组合键试图自动输入表格名称时，会产生错误消息"Configuration not available nor open"（配置不可用或打不开）。不过，我认为这里用"or"更合适些，看起来需要将这个窗口和某个Hibernate Console配置关联起来。行，这就够了，虽然在Mac OS X下界面有些不清楚，不过顶部的菜单似乎就是个用来解决问题的好办法。

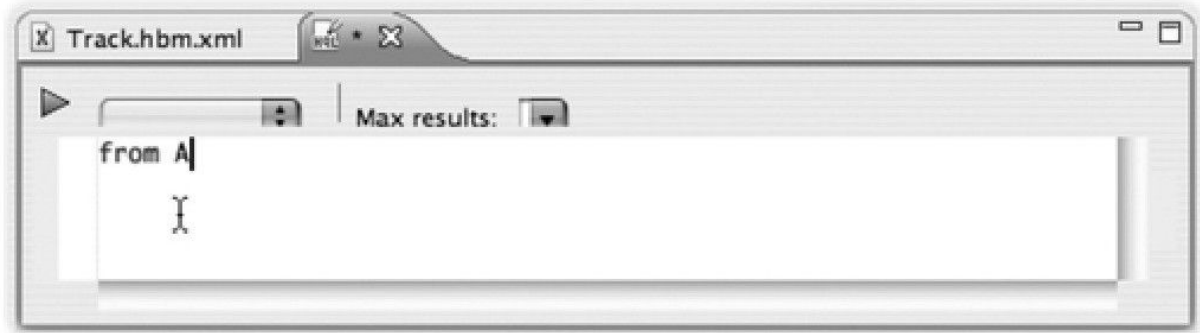


图 11-23 一个空白的、未经配置的HQL Editor视图

确实如此，配置菜单就在这里。在选择了菜单中的Hibernate ch 11配置项以后，就激活了编辑器的独特功能和自动完成的功能（如图11-24所示）。看起来可以将命名查询集中在一起，再粘贴到映射文档中，是吗？我们看看它还能做些什么事情。

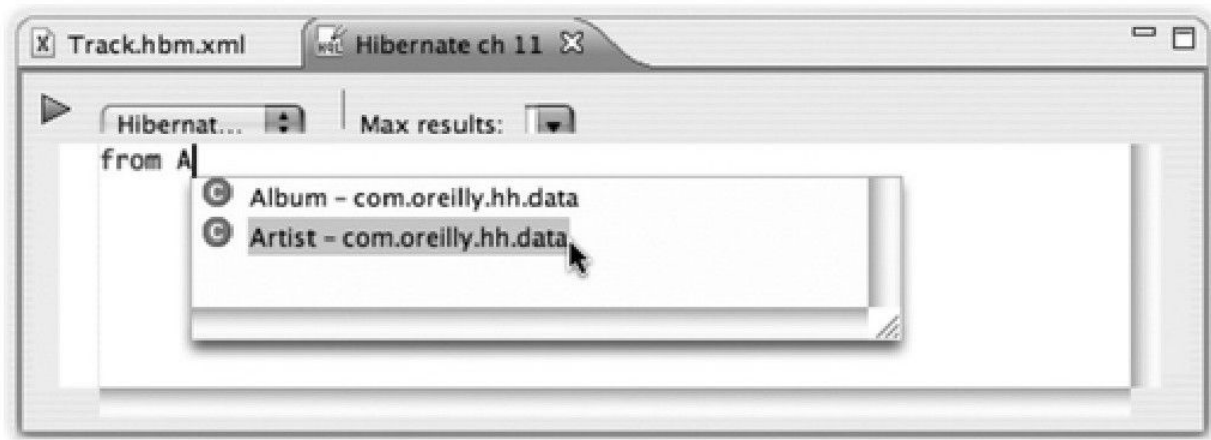


图 11-24 在连接到某个Hibernate Console配置以后的HQL Editor界面

在选择菜单项后，如果你也同样得到关于配置不可用的错误消息，这可能意味着Hibernate Tools还没有为它打开一个SessionFactory。你可以用鼠标右键点击Hibernate Configurations视图中的配置，再选择"Create SessionFactory"选项（如图11-25所示）。其他一些操作，比如向下深入（drilling）Configurations、Session Factory以及Database树节点的细节，似乎也需要一定的窍门。

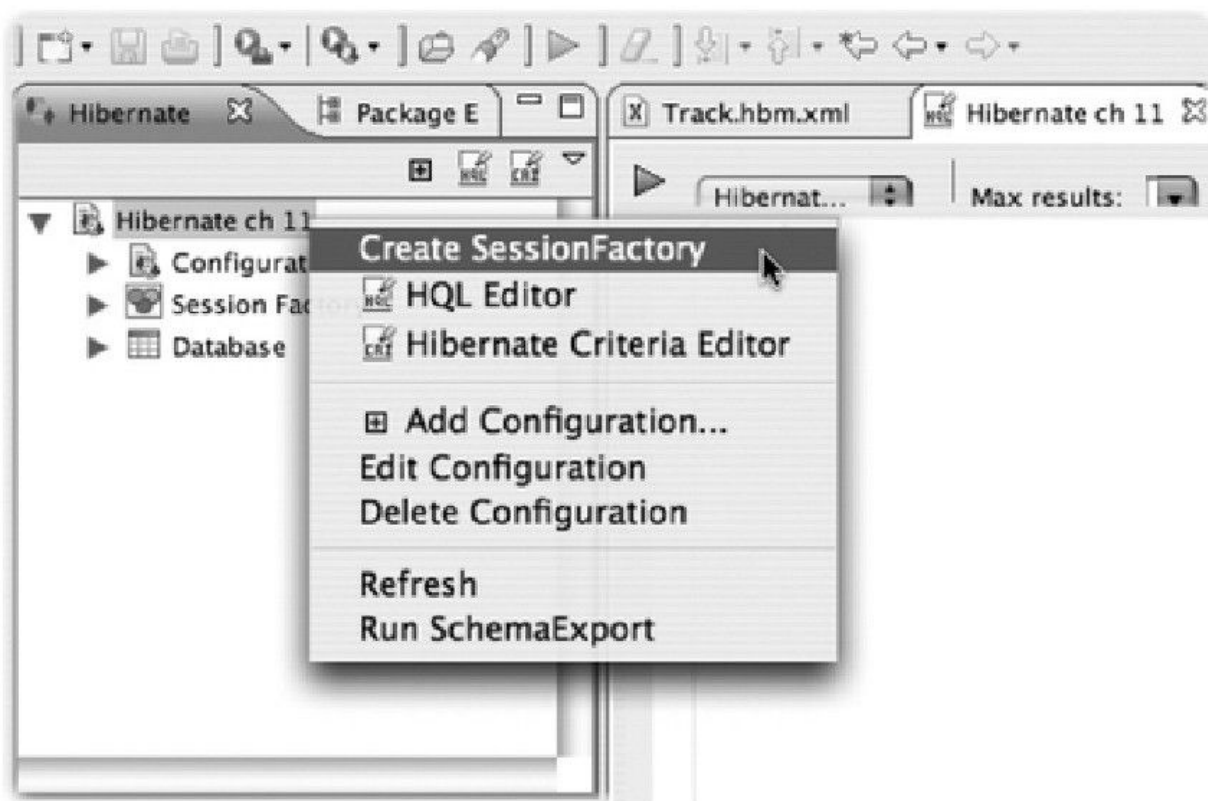


图 11-25 显式打开配置的SessionFactory

自动完成帮助对于编写查询确实有用（它可以完成属性名称、HQL关键字、函数名称等内容的辅助输入），不过这个工具真正强大的功能还要数它可以让你运行查询并查看结果，可以方便地验证查询和数据是否正确，或者只是为了更多地了解HQL和数据模型。点击编辑器左边顶部的较大的那个绿色“Run HQL”按钮，就可以运行HQL，并立即显示运行结果，如图11-26所示。

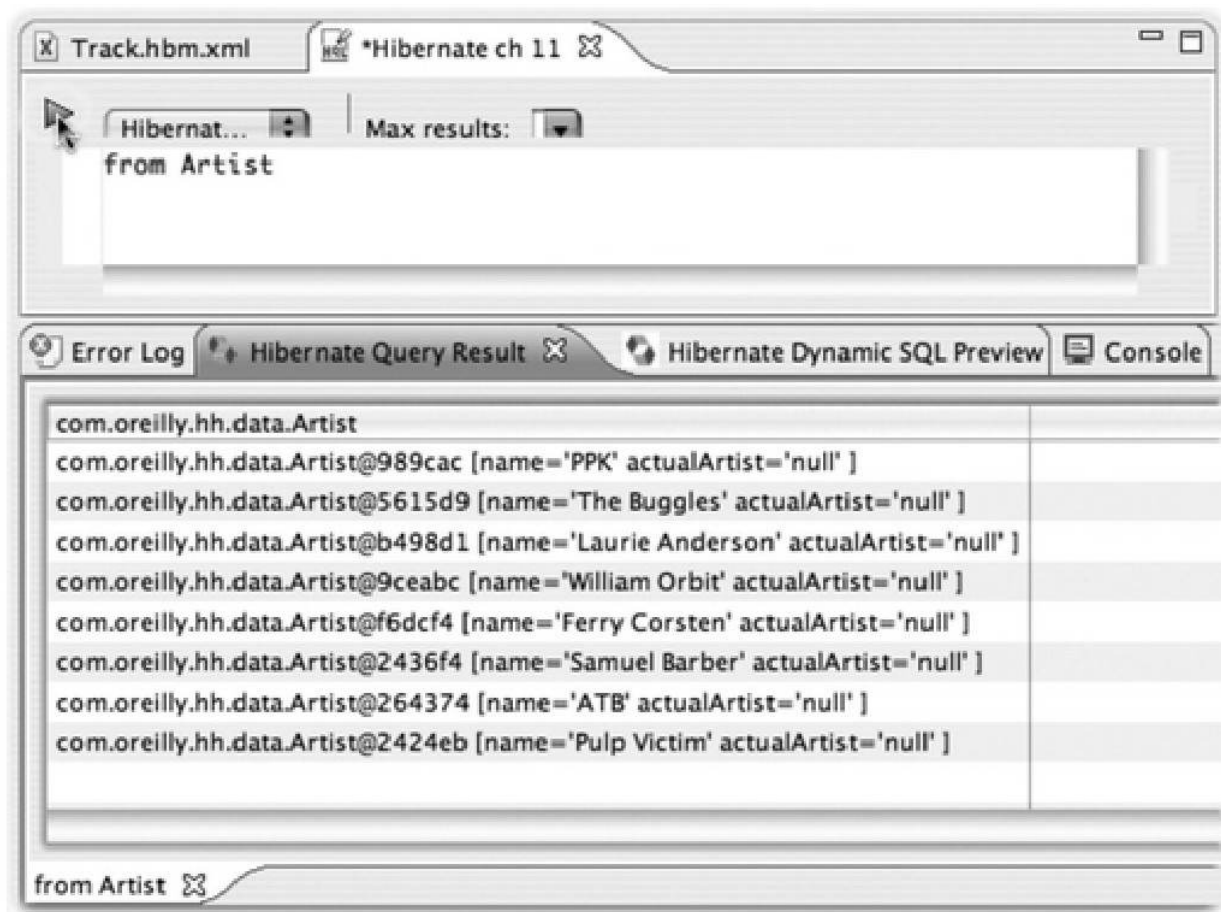


图 11-26 通过Run HQL按钮运行HQL，并查看结果

来吧，尝试一组。添加一些投影、排序以及聚合函数，这真是个好机会，可以真正实践一下第9章提到的那些查询功能！但是，在进入下一种编辑器以前，还有一两个窍门.....

如果你对大规模的数据库表执行查询，可能希望对结果返回的最多记录数量设置一定的限制，这时就可以使用编辑器顶部的第2个菜单。已经将数据加载到了内存中，如果没有的话，Eclipse则可能会有

崩溃的危险。当然，我们这本书中小儿科似的例子肯定不会有任何问题。

到目前为止，**Hibernate Configurations**视图下面的**Properties**（属性）视图的表现一般，当我们正在编辑映射文件时，它用于显示XML元素属性之类的东西，你以前在**Eclipse**中可能见过这些。点击**Hibernate Query Result**（**Hibernate**查询结果）视图中的一行，再看看会发生什么。图11-27是点击**Artist**查询结果中的**William Orbit**一行时的结果。

注意：多么可怕的原型、浏览、学习以及调试工具！

你可以查看选中结果对象的所有**Hibernate**属性，通过展开相应的三角形图标可以深入查看该对象的关联和集合。再一次，你应该自己尝试一下这些功能。（我们可能不用对此再提什么建议了.....）

The screenshot shows a 'Properties' window with a tree view on the left and a table of values on the right. The tree view is expanded to show the 'tracks' property of an album. The table lists various properties and their corresponding values.

Property	Value
▼ Identifier	
id	4
▼ Properties	
actualArtist	
name	William Orbit
▼ tracks	
▼ #0	
added	2007-09-24
▶ artists	
comments	
filePath	vol2/album972/track02.mp3
id	5
playTime	00:07:39
sourceMedia	CD
title	Adagio for Strings (ATB Remix)
volume	Volume[left=100, right=100]
▶ #1	

图 11-27 在Properties视图中浏览查询结果

你可能会问HQL编辑器右边的Query Parameters（查询参数）视图是做什么用的（如果没这样的问题，也可能是因为还没有显示这个视图，通过选择Window → Show View → Other菜单，再从弹出对话框的Hibernate部分选择Query Parameters，就可以打开这个窗口）。这个窗

口是用于处理查询中包含的命名参数的。图11-28显示了一个例子，我们从第3章粘贴了一个tracksNoLongerThan命名查询，接着发现点击Query Parameters窗口中的":P+"按钮，就可以生成查询中要用到的一个参数列表（这个例子只有一个参数）。我们必须将参数的Type设置为time（通过下拉列表选择），按照窗口列表下面提示的格式帮助信息，在Value列中输入一个时间值，再点击查询编辑器顶部左边的带有绿色箭头的"Run"（运行）按钮，就可以在底部的Hibernate Query Result窗口中看到希望的查询结果了。

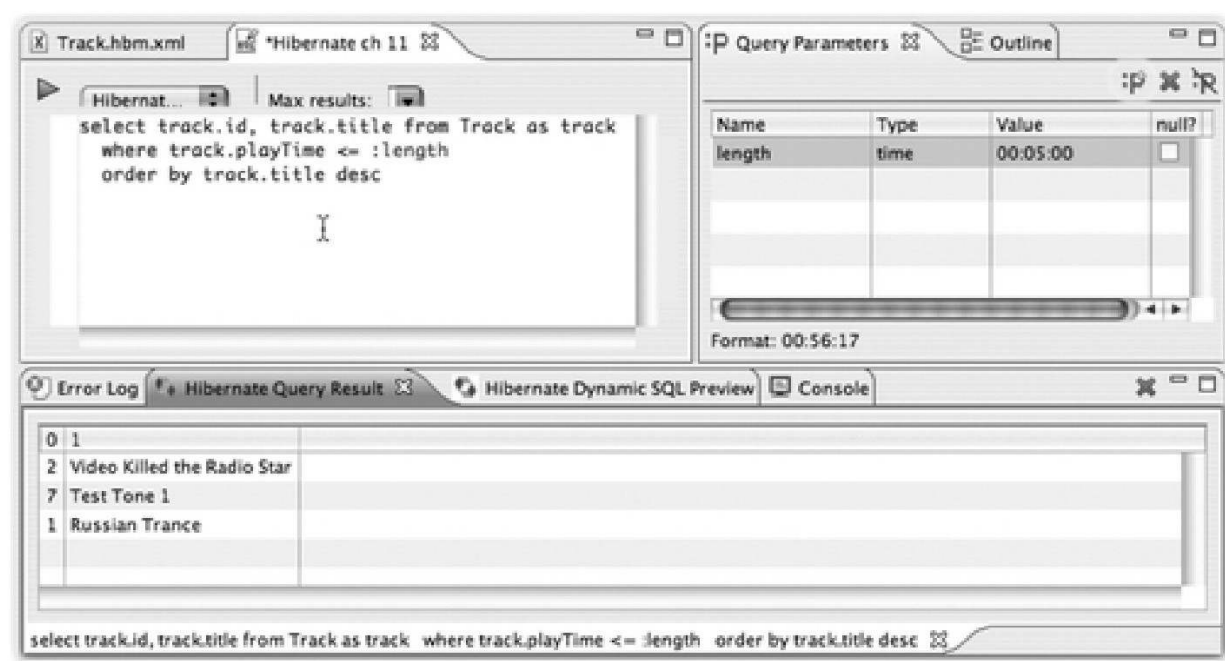


图 11-28 在Query Parameters窗口中设置命名参数

很难再想像出一个比这更简单的界面来试验HQL查询了！HQL查询触发的核心SQL语句可以在下面这个窗口中查看：Hibernate Dynamic SQL Preview（Hibernate动态SQL预览），在图11-28中它紧挨着查询结

果选项页；如果你在自己的Eclipse中看不到这个窗口，可以通过选择Window → Show View → Other菜单，再从弹出对话框的Hibernate部分选择Hibernate Dynamic SQL Preview，就可以打开这个窗口。在这个窗口中看到的SQL，就是我们在查询编辑器中输入的HQL在执行时使用的实际的SQL语句，如图11-29所示。

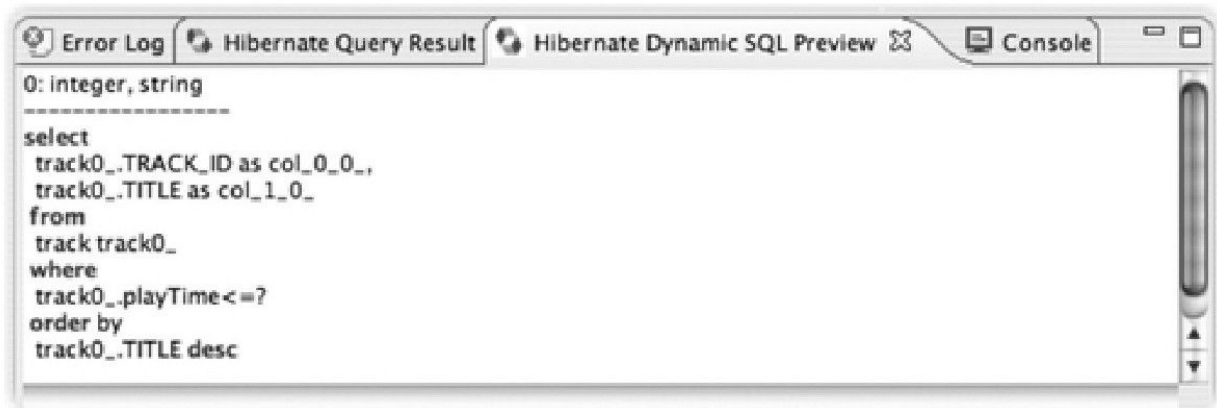


图 11-29 Dynamic SQL Preview窗口

这个窗口的内容非常有趣，因为它是真正动态的，正如其窗口标题所示。如果在HQL编辑器中编辑查询的同时，保持打开这个窗口，就可以看到你正在输入的HQL语句所对应的SQL语句，这样的效果相当直观。例如，考虑如图11-30所示的非常简单而且自然的HQL查询，以及该HQL生成的SQL语句。

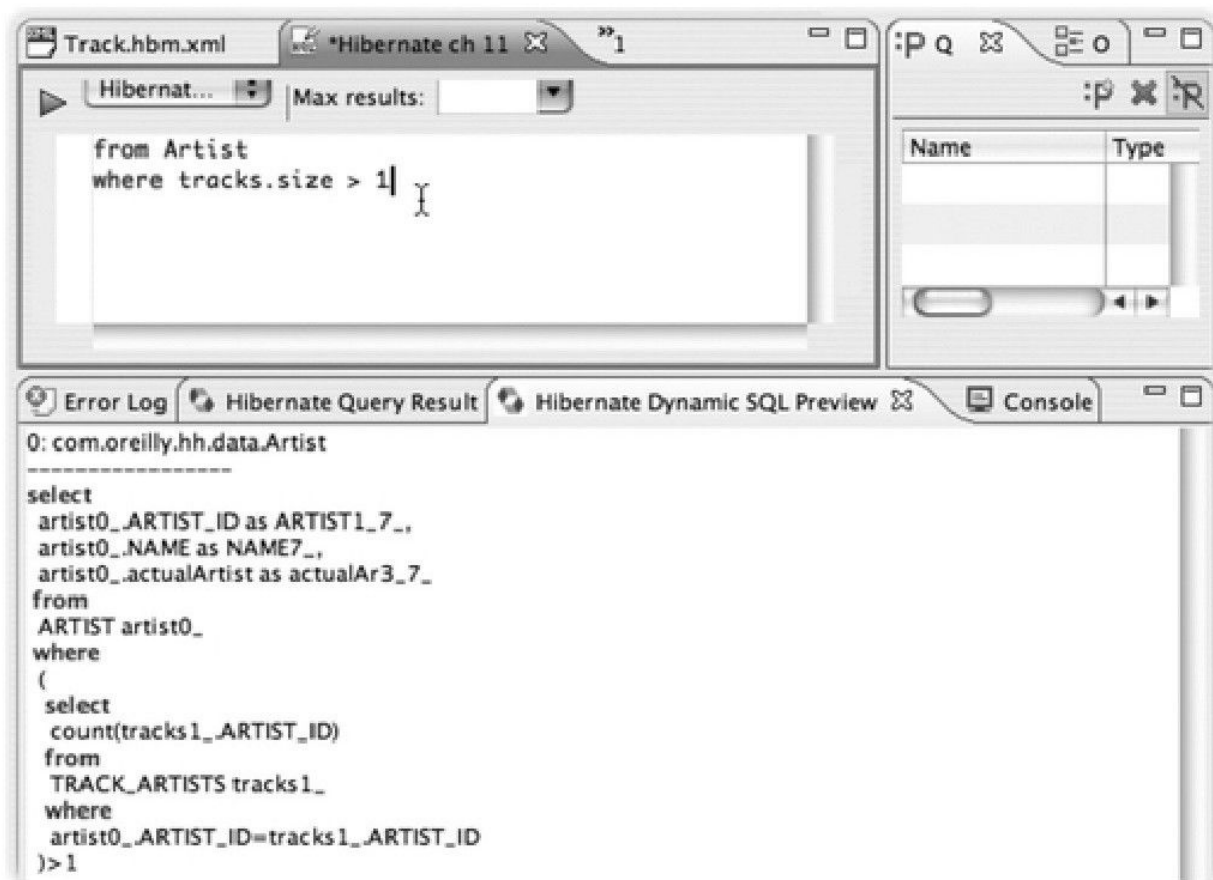


图 11-30 同时查看简洁的HQL与完整的SQL

就像我们的一位技术编辑指出的，这是在对SQL不十分了解的Java开发人员和数据库管理员之间进行交流的非常有价值的工具。

最后，Criteria Editor（条件查询编辑器）窗口，可以通过最后一个还没有介绍过的按钮访问，如图11-31所示，它的作用应该可以通过其名称猜得出来。它可以让你生成Criteria查询的原型，提供Java代码的自动完成（以及预定义的会话变量，以作为供查询使用的Hibernate Console配置的会话）。

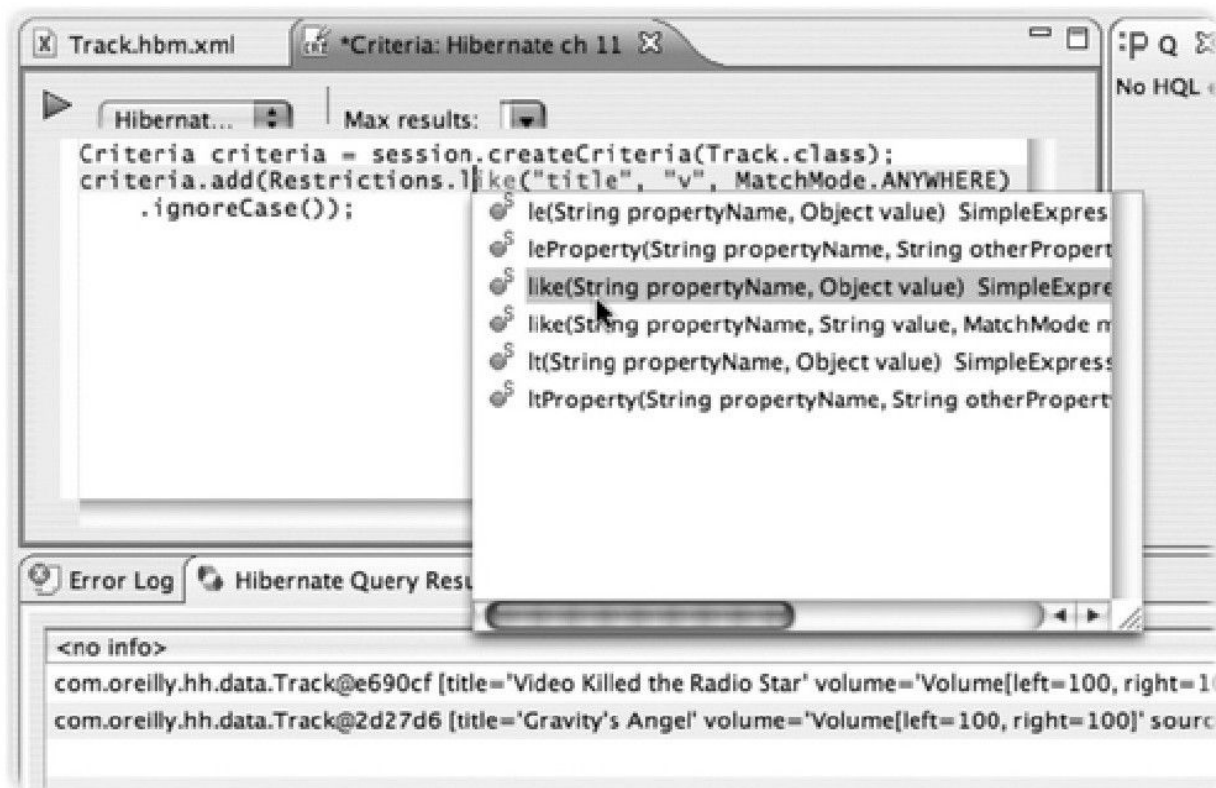


图 11-31 Hibernate Tools的Criteria Editor

如果你发现自动完成功能不能使用，记得检查Run按钮旁边的菜单中是否选择了有效的Hibernate Console配置，就像HQL Editor的使用一样。同时，也要确保Eclipse认为Criteria Editor窗口处于选中状态（其边框为蓝色）。有时我正准备要在Criteria Editor窗口中输入些东西时，而查询结果窗口的边框却是蓝色的，所以本来可以自动完成或编辑的键盘命令都无法正常使用。

其他

代码生成功能？这些也可以用，只是我们在前面还没有直接关注它们。在为特定的项目添加工具支持以前，它们就已经可以使用，不过在我们至少创建一个**Hibernate Console**配置以前，它们并不会做些什么。再仔细看看**Eclipse**工具条，就会发现**Hibernate Tools**还提供了一个新的菜单选项（如图11-32所示）。

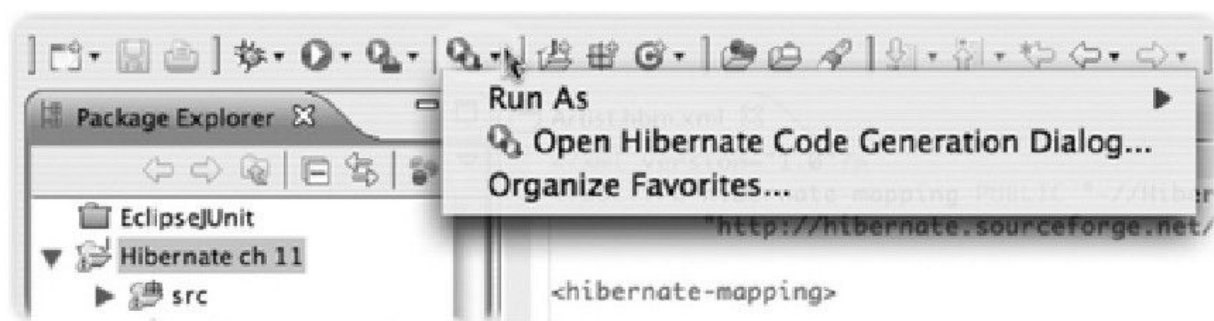


图 11-32 Hibernate Tools提供的代码生成菜单

这个新菜单隐藏得很隐蔽，需要费些时间才能看到它。这个菜单是访问**Hibernate Tools**用来建立和运行代码生成功能的图形界面的门户。

代码生成

让我们看看如何用Hibernate Tools实现原来在第2章中创建的codegen Ant构建目标完成的功能。为了避免翻书查找例子的麻烦，重新在例11-1中列出该构建目标的内容。

例11-1：重温代码生成的Ant构建任务

```
<!--Generate the java code for all mapping files in our source
tree-->
<target name="codegen"depends="usertypes"
description="Generate Java source from the O/R mapping files">
  <hibernatetool destdir="${source.root}">
    <configuration
configurationfile="${source.root}/hibernate.cfg.xml"/>
    <hbm2java jdk5="true"/>
  </hibernatetool>
</target>
```

为了在Eclipse中重新实现这一功能，我们先从图11-32所示的菜单中选择Open Hibernate Code Generation对话框。窗口界面如图11-33所示，像其他Eclipse工具一样，这个对话框也提供了让你选择命名配置的方法。

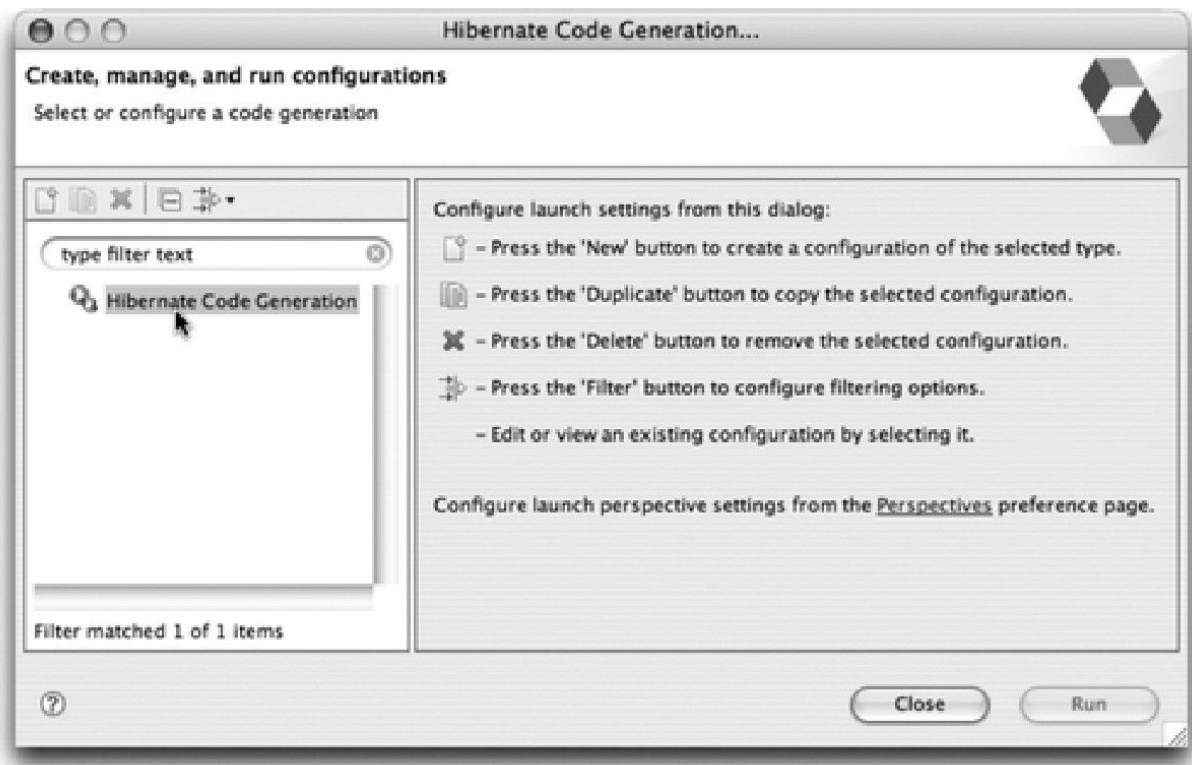


图 11-33 准备创建一个新的Hibernate代码生成配置

看起来为了激活创建新配置的功能，在点击"New"按钮以前需要先点击"Hibernate Code Generation"选项卡。

在激活创建新配置的功能后，就可以点击"New"按钮（这个按钮的图标是一个页面标志，在页面的右上角有个加号）来建立一个配置，以生成我们的数据对象。它会打开一个配置窗口，上面有很多选项，其中的一些选项如图11-34所示。

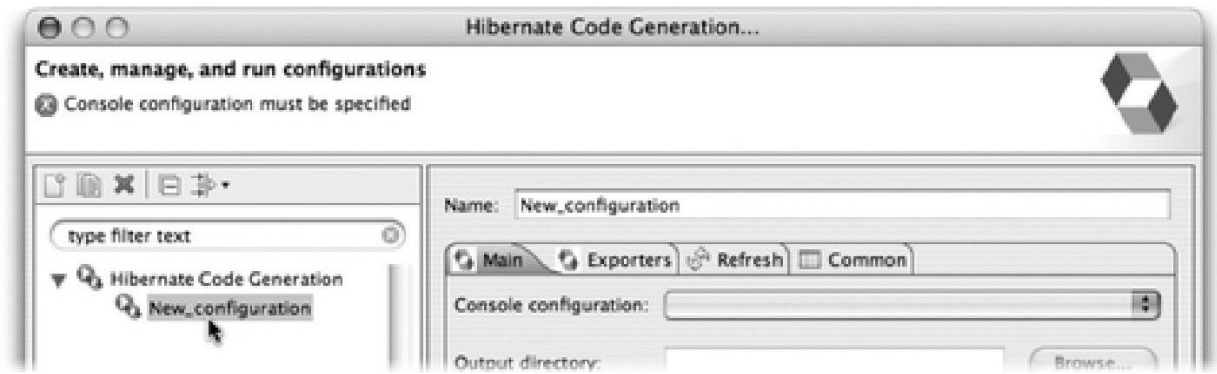


图 11-34 创建一个Hibernate代码生成配置

注意：当然，如果你在Eclipse工作空间中打开多个项目，在这个浏览对话框中就可以看到更多的选择根节点。

我们将新的配置命名为"Generate Ch 11 Model"，选择Hibernate Console配置，浏览打开项目的src目录（如图11-35所示），最终对话框的Main选项页界面应该如图11-36所示。



图 11-35 指定生成的代码的输出目录

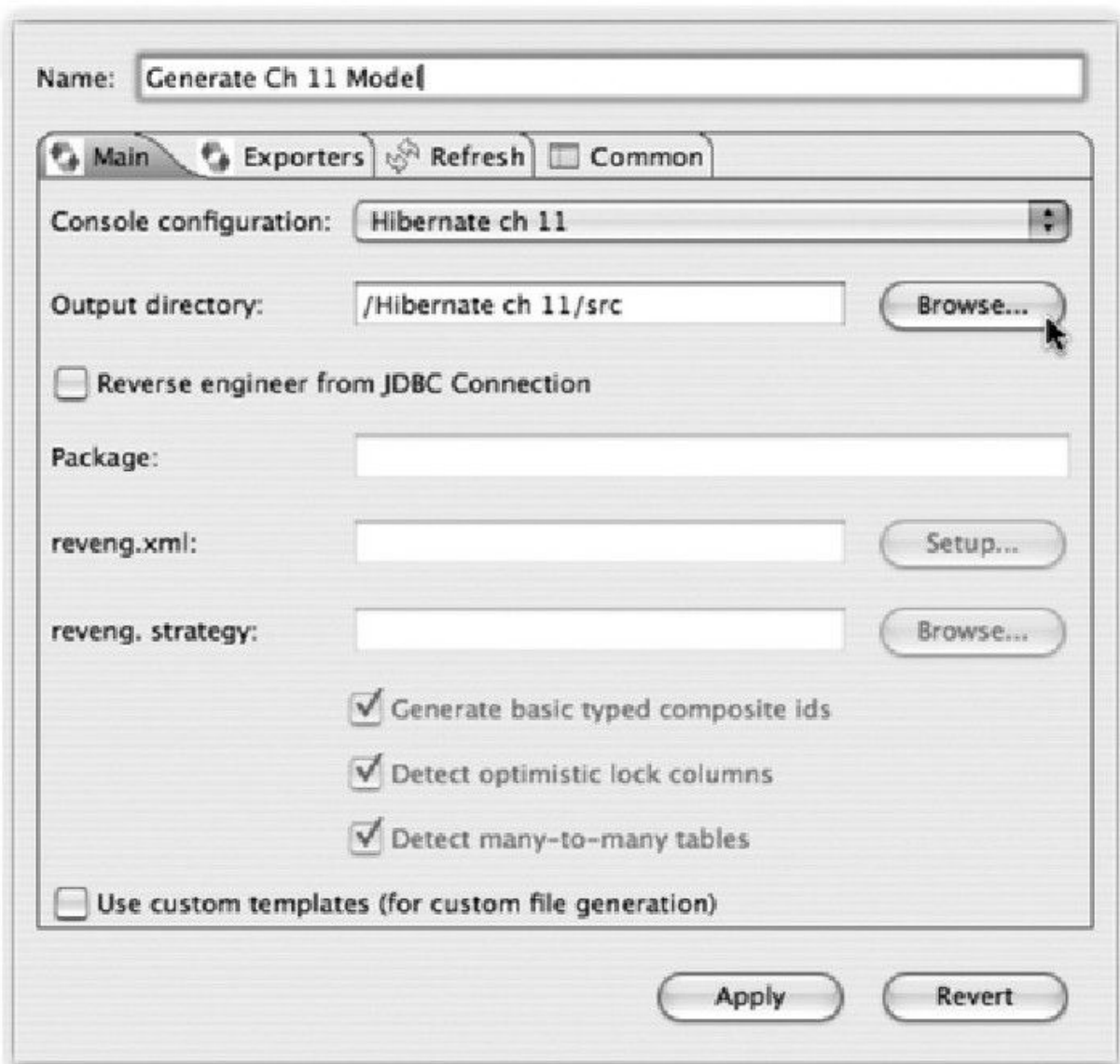


图 11-36 在Main选项页中设置代码生成的配置，以实现用codegen
Ant构建目标完成的功能

我们只需要关注Name，以及Main选项卡上的头两个选项，因为我们并不打算对现有数据做些奇特的反向工程处理（哪天需要将遗留下来的大型数据库模式转换到Java模型时，你可以自己再研究一下这个强大功能）。接着，我们再转移到Exporters（导出）选项卡。点击这个

选项卡，将打开如图11-37所示的程序界面，填写些信息，以再现原来的Ant构建目标的行为。

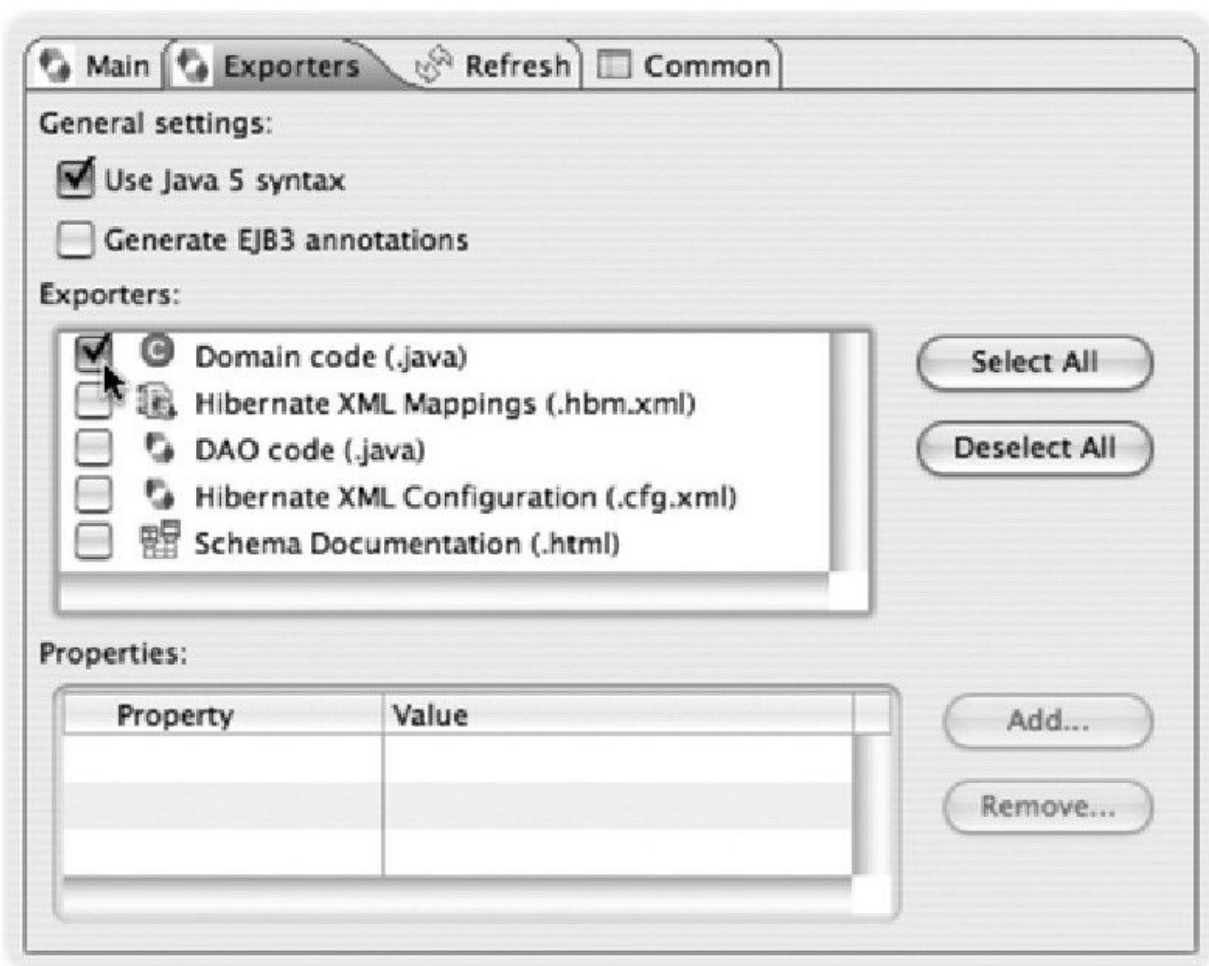


图 11-37 Exporters配置，以再现原来的codegen Ant构建目标

我们选中了Use Java 5 syntax复选框，就相当于在Ant构建任务中设置了jdk5=true属性；同时也选中Domain code（.java）导出选项。注意，虽然我们用的不多，还是有很多其他导出生成器可供我们用于创建映射文件、全局的Hibernate配置文件甚至是数据库模式的某种Web文档。此外，也可以创建某种数据访问对象（DAO代码），以标准化、

方便的方式来简化对模型对象的加载和处理。虽然与这些选项相关的主题已经超出本书的讨论范围，不过你可以自己研究它们的使用方法，至少应该先记下来，以备不时之需。

注意：提供GUI的目的是为了解决什么……至少它们涵盖了常见的应用案例。

可能你会问选项卡底部的**Properties**部分是做什么用的。它基本上用于让你设置那些不能用图形界面表示的每个**Exporters**（导出器）参数。点击任何一个**Exporters**（列表中的名字，而不是复选框），就会显示手工为那个生成器设置的所有属性，同时也激活**"Add"**和**"Remove"**按钮，以便可以编辑各个属性。对于有什么属性，以及它们的作用，可以参考生成器的文档。

对于我们当前的任务，就是要重新实现以前用**build.xml**文件实现的功能，这个文件内部的那些配置就是正确的设置。事实上，这些也正是我们真正需要设置的。不过，在这里看一些**Eclipse**特定的选项，也是有价值的。**Refresh**（刷新）选项卡（如图11-38所示）可以让你控制在运行**Exporters**以后，**Eclipse**应该自动刷新哪些资源。

图中我们选择刷新生成代码所属于的那个项目。似乎这样符合我们正在进行的操作。



图 11-38 代码生成后的刷新选项

最后，Common（通用）选项卡上的默认设置看起来就不错了，不需要修改，点击"Apply"（应用）按钮。

我们刚才的配置将出现在左边的列表中（如图11-39所示）。这时我们可以先选中它，再点击"Run"按钮（如图11-34的底部右边所示）。或者，如果我们认为可能需要经常运行它的话，可以将它设置为工具条菜单上的一个快捷按钮，这样它就可以出现在菜单的顶部了。从现在起，我们只要从这个对话框中选择相应的配置，再点击"Run"按钮，就可以测试运行了。

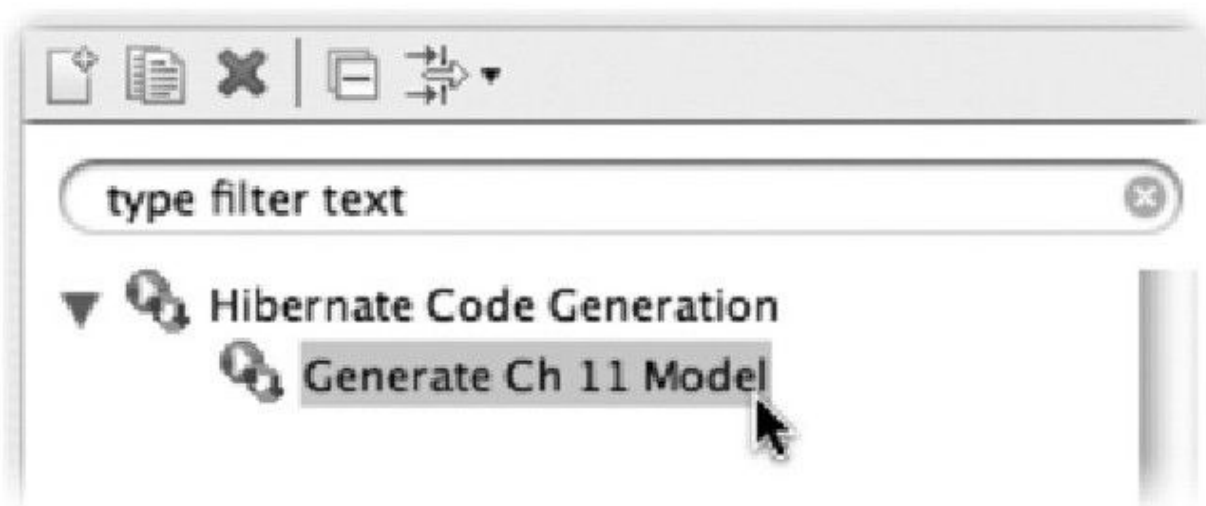


图 11-39 我们的代码生成配置现在可以运行了

运行它并不会产生什么特别的结果，只是Eclipse窗口底部右边的状态栏会简单地显示正在进行的后台活动（我们保持Common选项卡中的Launch in background配置选项为选中状态，这正是我们想要的结果）。要看看它是否做了些什么，可以找到生成的类（例如Album.java）来看看。例11-2中的代码演示了由Hibernate Tools生成的Album类源代码的开始部分。

例11-2: Eclipse中的Hibernate Tools生成的Album类的源代码

```
package com.oreilly.hh.data;
//Generated Jan 5, 2008 6: 36: 04 PM by Hibernate Tools 3.2.0.CR1
import java.util.ArrayList;
import java.util.Date;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
/**
 *Represents an album in the music database, an organized list of
 tracks.
```

```

* @author Jim Elliott (with help from Hibernate)
*
*/
public class Album implements java.io.Serializable {
    private int id;
    private String title;
    private Integer numDiscs;
    private Set<Artist> artists = new HashSet<Artist> (0);
    .....

```

源文件中的时间戳和工具版本号都表明这个类是最新生成的（想到我们正在使用的**Maven for the Ant tasks**中提供的**3.2.0.b9**版本，谁知道最新版本进入**Maven**仓库的延迟会有好的一方面呢？）。另一方面，这一结果看起来与**Ant**生成的结果类似。并不太令人振奋，或许我们已经预见到了这些。

那还剩下什么要我们去做？像这样内建到**Eclipse**中的工具可以极大地节省项目的设计和开发周期。能够将查询和数据库模式辅助自动完成、属性浏览以及实时**SQL**显示等集成在一起，这本身就是一种理解数据和模型对象的很好方法。本书前面为测试各种查询而需要编写单独的代码示例，再编译、运行、调整它们，与这种繁琐的方法相比，图形化工具无疑更加快速、更加方便。你可以快速地测试许多查询的变化，而且不需要繁琐的模板代码（无论如何，使用**Hibernate**后，这样的代码已经没有什么了）来建立运行环境。

映射图表

在本书大部分的写作时间内，**Hibernate Tools**一直都是Beta版本，还不支持本节最后介绍的这个功能。幸好，就在本书交付印刷前夕，发布了3.2的最终版本，这一版本支持映射图表。对象和数据模型之间的图形视图对理解它们之间的关系很有帮助，现在Eclipse中就可以生成这样的图表了。为了创建图表，先在**Hibernate Configurations**视图中选择一个映射类，打开它的上下文关联菜单（用鼠标右键点击元素或在按住**Control**键的同时点击元素，如图11-40所示），再选择**Open Mapping Diagram**。

这会生成一个类似图11-41的新视图。为了让视图适合页面的大小，这里选择了一个简单的类，但如果使用大屏幕，你可以滚动视图，就能够查看对象之间复杂的关联关系，信息量非常大。如果你不喜欢图表中各元素的位置，可以随意拖动它们到任何地方；还可以使用图表中的上下文关联菜单来打开你感兴趣的源文件和映射文档。

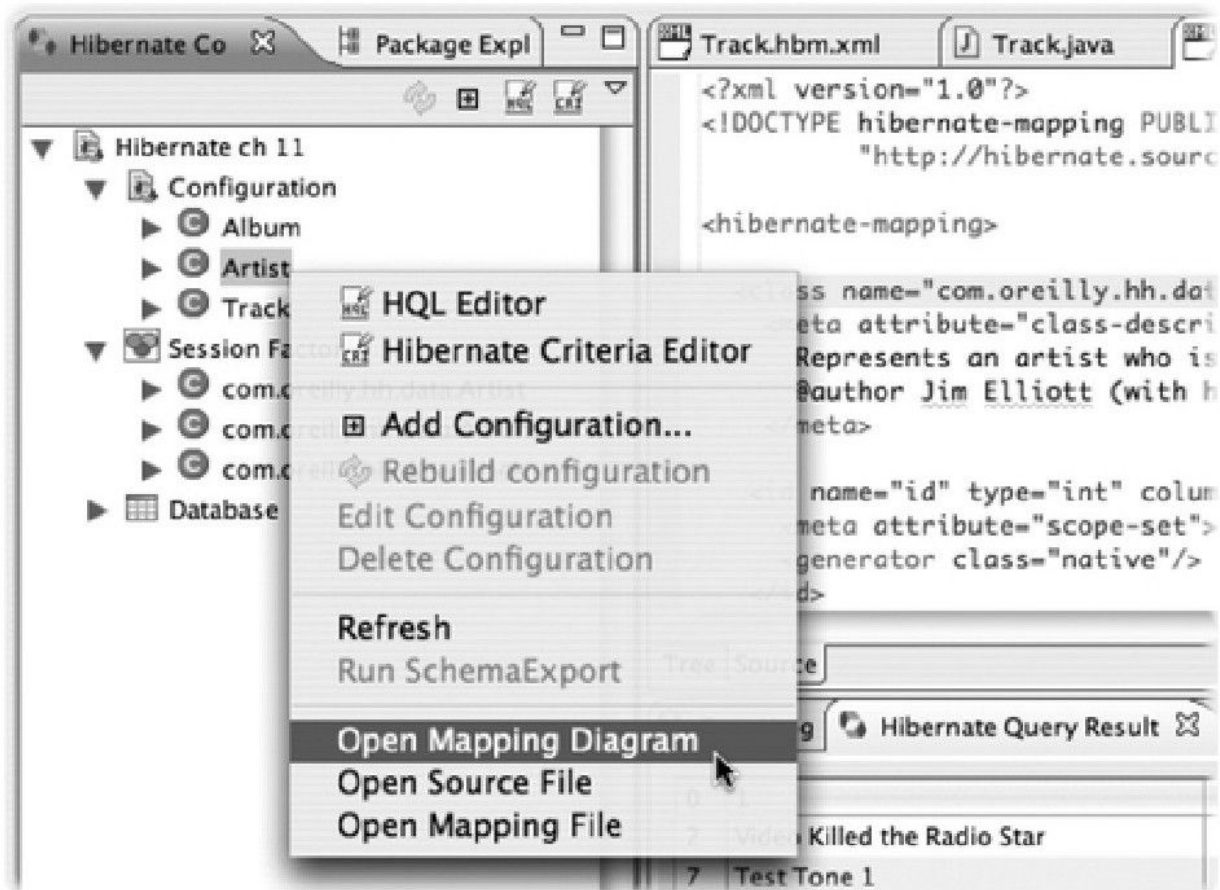


图 11-40 打开一个映射对象的模型图表

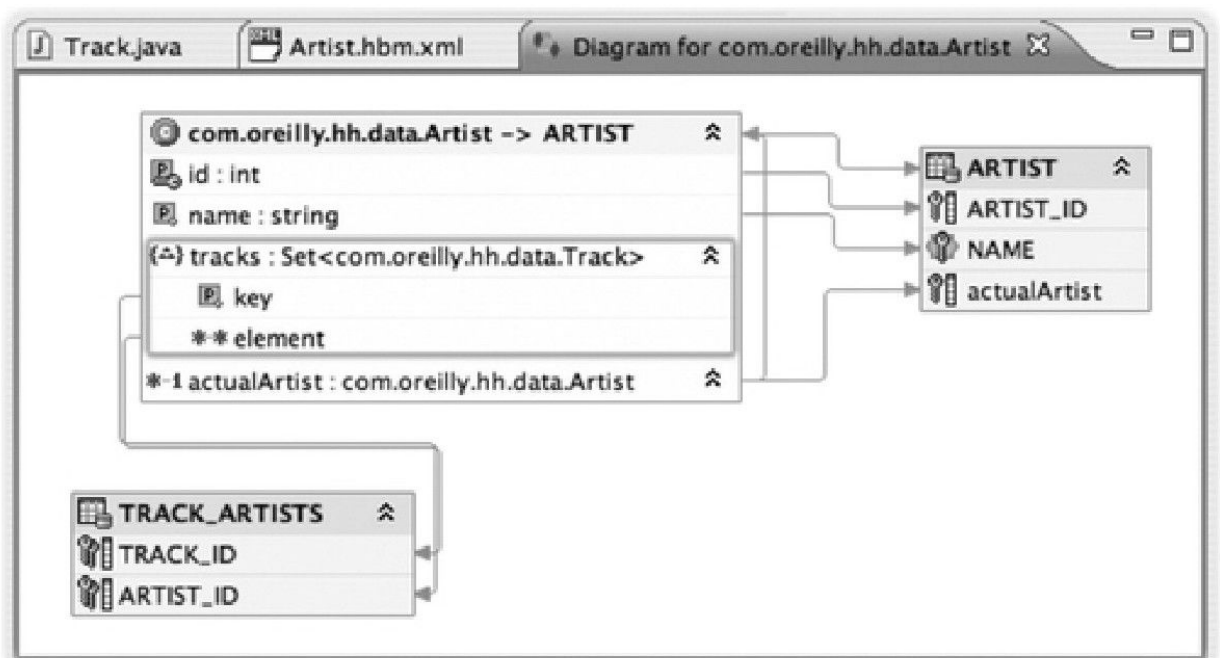


图 11-41 类的映射图表

明显地，可感知数据库模式（**schema-aware**）的XML和查询编辑，用于构建映射和**Hibernate**配置的GUI选项，这些相当有用的功能是将**Eclipse**作为开发首选的工具而获得的好处。用图表来帮助实现数据的可视化和理解，真是为**Eclipse**锦上添花。而且，能够与**Hibernate**会话动态地进行交互，点击一下按钮就可以生成代码，在**IDE**中就可以运行查询，甚至不用建立**Ant**构建文件就可以完成很多用其他办法无法实现的功能。如果你正在考虑这些文字介绍，对如何才能不需要很麻烦地手工将**Maven**资源添加到类路径中而感兴趣，下一章将会为你独辟蹊径。

第12章 Maven进阶

如果从O'Reilly的网站下载我们的示例代码，你会发现每一章的示例目录都包含一个神秘的pom.xml文件，我们还没有介绍过这个文件。pom.xml是Apache Maven这个构建工具的配置文件，现在这个工具已经广泛用于取代Apache Ant。对Maven完整的介绍不在本书讨论的范围以内，不过，我们觉得这里介绍些有关如何随Hibernate一起安装和使用Maven，这些应该够用了。本章旨在为Maven提供最简洁的介绍，重点说明Maven和Hibernate3插件的使用，顺便也会介绍一些Maven的核心概念。

什么是Maven

Maven是一种声明式（declarative）的构建工具，它不是定义一套用于构建项目的过程步骤，而是用保存在pom.xml文件中的Project Object Model（POM，项目对象模型）来描述项目。担当重任的Maven插件，它们知道如何读取POM文件和完成任务。例如，默认的Maven插件可以编译代码、创建JAR文件、组装WAR文件、创建Web网站、为JAR文件生成数字签名（sign）、计算代码质量度量（code metrics）、执行单元测试、读取Hibernate映射文件等。使用Maven，你所有需要做的就是告诉它源代码在哪里，需要依赖什么文件，

Maven足够聪明，它会明白接下来应该做什么。如果不用**Maven**，而是使用像**Apache Ant**这样的工具，就得定义一个显式的构建过程。如前所述，**Maven**采用声明式的方法来构建和测试一个项目，作为**Ant**的替代品，它很快就受到了欢迎。

注意：**Maven**可以为你节约很多时间，但是你要去学习和习惯它。

获得方便的同时也需要付出一定的代价。**Maven**之所以知道如何处理你的项目的源代码、配置以及单元测试，是因为它做了一些假设。首先，**Maven**假设你的项目采用的是标准布局和构建生命周期。所以，在使用**Maven**之前，我们希望你知道**Maven**的这些核心假设。一个假设就是对项目的定义。在**Maven**中，一个project是由源代码和资源组成的，每个资源与一个artifact对应。这个artifact可以是类似JAR或WAR之类的文件，不过，需要知道的一个重点就是一个项目只有一个artifact。另一个假设是（大多数情况下）你应该使用标准的目录布局。在pom.xml文件中可以对**Maven**的大部分默认设置进行修改。如果你不想将源代码保存在src/main/java目录中，则可以在pom.xml中指定另一个目录。

注意：如果你不喜欢**Maven**的约定，可以考虑换个工具。

如果你喜欢Maven，但手边的几个项目确实不能完全满足Maven假设的要求，也不必放弃Ant。事实上，Maven在现有的Ant构建文件中对它进行调用也提供了方便的机制。除了本章少数几个示例以外，本书的大部分示例就以Ant构建文件作为基础。如果你下载随书的示例代码，就会发现本章特殊的示例具有与其他章节不同的目录布局。我们研究一下这种目录结构，也就是Maven中所谓的标准目录布局（Standard Directory Layout）。

有关Maven更详细的介绍，可以阅读Sonatype的《Maven: The Definitive Guide》（[\[1\]](#)）。

Maven的标准目录布局

理想的构建工具应该能够自动知道如何处理一套Java源文件。这样的系统将会具有一定的内建智能，可以让你简单地编写一定的源代码，再将简单的文件放在项目根（root）目录下，运行构建工具，再回到项目目录。构建工具会认为你的项目包含一定的Java源代码，你想根据它们而生成一个JAR文件。如果你想生成的是WAR文件，就应该提供一些提示，以修改构建工具的默认行为。

这种思想已经由Maven实现了，至少实现了一部分，因为Maven提供了一套严格的约定，消除了大部分常见配置的需要。这就是所谓

的“约定优于配置”（convention over configuration），在过去几年中这种思想在Ruby on Rails之类的框架中变得非常流行，它让你即便是编写非常复杂的Web应用程序也不必为不计其数的配置文件而费心（第14章介绍的Stripes也采用了Java业界的类似方法）。约定优于配置，这也是为什么Apple笔记本电脑总是可以开箱即用（out of the box）的原因，也是为什么在驾驶汽车前你不用阅读它的使用手册就知道油门在右边而刹车在左边的原因。通过同样的策略（如果你已经遵守了约定），Maven就会自动知道你的源代码、单元测试、网站文档以及配置文件都在什么地方，而不用浪费你宝贵的时间亲自告诉Maven这些东西都放在哪儿。例12-1显示了Maven的项目目录布局的大致样子。

例12-1: Maven的标准目录布局

`pom.xml`

每个项目都必须有一个pom.xml文件（即一个POM）。本章后面的第12.6节将详细介绍POM文件。

`src/`

在这个子目录中保存各种源文件。

`src/main/java/`

将要包含在最终artifact（JAR文件、WAR文件等）中的Java源代码位于这个目录。

`src/main/resources/`

这个目录包含项目需要的各种资源：非Java源代码、类似log4j.properties的东西、hibernate.cfg.xml、本地化（localization）文件以及需要发布的Hibernate映射文件。

`src/test/java/`

保存Unit测试代码（TestNG或JUnit测试）。

`src/test/resources/`

保存只供测试使用的资源。

`src/site/`

网站文档保存目录。

`target/`

构建过程生成的任何东西，可以是源代码、字节码或最终的产品，都将保存在target目录中。因为target中的文件是构建过程的产物，所以可以随意删除，不应该包括在版本控制（version control）中。

还有一些其他目录，比如src/main/webapp、src/main/config、src/main/assembly以及src/main/filters，这里不对它们进行过多的介绍，在充分利用Maven创建复杂的应用程序时，这些目录都有各自不同的用途。例如，如果我们要生成一个WAR文件，则src/main/webapp将是Web应用程序的文档根目录。如果我们为命令行应用程序创建一个自定义的文档包，则src/main/assembly目前用于保存我们将要使用的装配描述符（assembly descriptor）。例12-1中列举的那些目录将出现在每个Maven项目中，它们是Maven项目最低的公共约定标准。

虽然这些看起来似乎是相对简单的想法，但业界目前还没有对Java项目的标准目录布局之类的事达成一致。事实上，反对使用Maven最常见的一个理由就是人们不喜欢Maven约定的目录结构。其实你可以容易地自定义这种目录结构，如果你看看本书其他章节的示例代码，就会发现我们采用的也是自己的目录布局结构。这种方法也有一些缺点，例如，由于使用自定义目录布局而引起的微妙问题，有几章示例就与Maven Hibernate3插件并不完全兼容。所以我向你再次警告，尽可能不要试图让Maven适应你的项目的自定义目录布局，而是在自定义之前调整你的项目以适应Maven的标准目录布局。不过有些简单的区别也不会给Maven造成任何问题，比如将资源保存到src/main/java目录下，或者配置Maven，让它把源代码保存到src/java目录而不是src/main/java目录下。就像Maven对源代码有一定的标准一样，它对工具生成的源代码也有一定的约定（这些代码文件最终位于

target/generated-source目录下，它们包含在"compile source roots"列表中，所以它们将与你自己编写的代码一起进行编译）。

注意：本章的主题或者是“接受Maven的约定”，或者是“你将被同化。抵抗是没用的”。

如果当你读完本节后会想“嗯，我们有更好的目录布局”，非常不幸，Maven不适合你了。如果你想试图调整一些Maven的核心假设，可能要花费好些天来对付一系列莫名其妙的问题。最后，Maven跑不起来了，你将开始大声地诅咒Maven，接着在博客上发帖子说你有多么多么地讨厌它，这一切都因为你不愿意接受Maven基本的假设。可能你还没有注意到，本章开始的几节都在尽可能地为你节省时间。我可以证明Maven为了节省了无数的时间和精力，从效率的角度来说，给了我意外的收获。但是我也看到有些人使用Maven的方向就是错误的，最终除了对Maven的不满以外，项目上一无所获。适应Maven，而不是让Maven适应你。这些已经接近事实，但如果你一开始就质疑这些事实的话，受伤的就只能是你自己。

[1] <http://www.sonatype.com/book/index.html>.

安装Maven

Maven的二进制发行版本可以从<http://maven.apache.org/download.html>下载，下载适合你使用的格式的当前最新版本，为它选择一个合适的保存位置，并在那里解开压缩包。如果你将文档解压到类似/usr/local/maven-2.0.8这样的目录中，则可能还应该创建一个到这个目录的符号链接以方便使用，这样，在更新到新版本时也不必修改环境配置：

```
/usr/local%ln -s maven-2.0.8 maven
/usr/local%export M2_HOME=/usr/local/maven
/usr/local%export PATH=${PATH}: ${M2_HOME}/bin
```

在安装好Maven后，还需要做两件事情，才能让Maven正常运行。你需要将Maven的bin目录（在这个例子中是/usr/local/maven/bin）添加到命令行路径中。还需要设置环境变量M2_HOME，让它指向Maven安装的顶级目录（在这个例子中是/usr/local/maven）。有关如何在各种不同的操作系统中执行这些安装步骤的细节，可以参阅《Maven: The Definitive Guide》一书。

项目的构建、测试以及运行

假设你已经从本书的网站（[\[1\]](#)）下载好了示例代码，这时应该转到ch12示例目录，运行`mvn test`命令。运行结果如图12-2所示。

例12-2：告诉Maven测试我们的项目

```
$mvn test
[INFO]Scanning for projects.....❶
[INFO]-----
-----
---
[INFO]Building Harnessing Hibernate: Chapter Twelve: Maven
[INFO]
task-segment: [test]
[INFO]-----
-----
---
[INFO][resources: resources]
[INFO]Using default encoding to copy filtered resources.
[INFO][compiler: compile]❷
[INFO]Compiling 9 source files to~\examples\ch12\target\classes
[INFO]Preparing hibernate3: hbm2ddl
[WARNING]Removing: hbm2ddl from forked lifecycle, to prevent
recursive
invocation.
[INFO][resources: resources]
[INFO]Using default encoding to copy filtered resources.
[INFO][hibernate3: hbm2ddl{execution: generate-ddl}]❸
[INFO]Configuration XML file loaded:
~/examples/ch12/src/main/resources/hibernate.cfg.xml
20: 16: 05, 580 INFO org.hibernate.cfg.annotations.Version-
Hibernate
Annotations 3.2.0.GA
20: 16: 05, 595 INFO org.hibernate.cfg.Environment-Hibernate
3.2.0.cr5
20: 16: 05, 598 INFO org.hibernate.cfg.Environment-
hibernate.properties
not found
```

```

20: 16: 05, 599 INFO org.hibernate.cfg.Environment-Bytecode
provider name
: cglib
20: 16: 05, 603 INFO org.hibernate.cfg.Environment-using JDK 1.4
java.sql
.Timestamp handling
[INFO]Configuration XML file loaded:
~/examples/ch12/src/main/resources/hibernate.cfg.xml
20: 16: 05, 684 INFO org.hibernate.cfg.Configuration-configuring
from url:
~/examples/ch12/src/main/resources/hibernate.cfg.xml
20: 16: 05, 808 INFO org.hibernate.cfg.Configuration-Configured
SessionFactory:
null
(schema export omitted)
20: 16: 07, 172 INFO org.hibernate.tool.hbm2ddl.SchemaExport-
schema export
complete
20: 16: 07, 173 INFO
org.hibernate.connection.DriverManagerConnectionProvider
-cleaning up connection pool: jdbc: hsqldb: data/music
[INFO][resources: testResources]
[INFO]Using default encoding to copy filtered resources.
[INFO][compiler: testCompile]④
[INFO]Compiling 2 source files to~\examples\ch12\target\test-
classes
20: 16: 08, 194 INFO
org.hibernate.connection.DriverManagerConnectionProvider
-cleaning up connection pool: jdbc: hsqldb: data/music
[INFO][surefire: test]⑤
[INFO]Surefire report directory: ~
\examples\ch12\target\surefire-reports
-----
T E S T S
-----
Running com.oreilly.hh.ArtistTest
Hibernate: insert into ARTIST (ARTIST_ID, actualArtist_ARTIST_ID,
NAME)
values (null, ?, ?)
Hibernate: call identity ()
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
1.345 sec
Results:
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
这个命令进行了很多活动，我们来看看刚才发生了什么：

```

❶首先，`mvn test`是什么意思？在命令行中，我们调用Maven，并向它传递Maven构建生命周期（**Maven Build Lifecycle**）中目标阶段的名称。Maven构建生命周期是Maven在构建项目时经历的一组阶段。在生命周期的各个阶段中，需要为了执行而注册不同的插件：在编译阶段要对代码进行编译，在测试阶段要执行各种测试，在打包阶段要创建JAR包。有关Maven构建生命周期各阶段的完整列表，可以参阅本章后面的“Maven构建的生命周期”一节。

❷要到达测试阶段，必须先通过编译阶段。在编译阶段需要注册并运行Maven Compiler插件。Maven插件是由不同的目标（goal）组成的，我们可以看到执行了编译目标，编译了9个源文件，生成的字节码文件放到了`target/classes`中。我们如何判断编译器插件已经执行了编译目标？`[compiler: compile]`告诉我们这一信息，在Maven的输出中，你会看到很多类似的信息。冒号之前的字符串部分是插件标识符（**plugin identifier**），之后是正在被执行的目标标识符（**goal identifier**）。注意，`compiler: compile`下面几行有一条警告性质的日志语句，目前你可以放心地忽略它。

❸接着，我们可以看到正在执行的是Hibernate3插件的`hbm2ddl`目标。这个插件不是默认插件，必须在Maven中明确地添加这个插件，不过，不必为它提供很多配置。这个插件假设`hibernate.cfg.xml`位于

src/main/resources 目录中。我们将在后面的“使用Maven Hibernate3插件”一节中详细地介绍Hibernate3 Maven插件。

④接着，编译单元测试。编译器插件有一个名为testCompile的目标。编译好的测试字节码保存在target/test-classes 目录中。

⑤最后，Surefire插件执行test目标，在生成的测试类中寻找扩展自JUnit的TestCase类的测试类。在这个例子中，我们编写了一个简单的JUnit测试，向ARTIST表插入一个测试值。

Maven刚才在幕后的操作就是编译源代码、创建一个HSQLDB数据库、编译单元测试、运行一个单元测试以便在数据库中插入一行数据。如果你想重复该示例，删除数据目录或单元测试将导致向数据库中插入一行数据时会失败，因为这样会违反在艺人名称上施加的惟一性约束限制条件。为了清除掉整个项目，应该运行mvn clean命令：

```
$mvn clean
[INFO]Scanning for projects.....
[INFO]-----
-----
---
[INFO]Building Harnessing Hibernate: Chapter Twelve: Maven
[INFO]
task-segment: [clean]
[INFO]-----
-----
---
[INFO][clean: clean]
[INFO]Deleting directory~\examples\ch12\target
[INFO]Deleting directory~\examples\ch12\target\classes
[INFO]Deleting directory~\examples\ch12\target\test-classes
[INFO]Deleting directory~\examples\ch12\target\site
```

如果你打算在其他项目中也利用这个项目的类，你可能希望生成一个JAR文件，并将其保存在类路径中。为了生成一个JAR，应该运行`mvn package`命令：

```
$mvn package
[INFO]Scanning for projects.....
[INFO]-----
-----
---
[INFO]Building Harnessing Hibernate: Chapter Twelve: Maven
[INFO]task-segment: [package]
[INFO]-----
-----
---
.....skipping output.....
[INFO][jar: jar]
[INFO]Building jar: ~\examples\ch12\target\hib-dev-ch12-2.0-
SNAPSHOT.jar
```

我们对这个示例的输出进行了大量删减，因为它与前面的`mvn test`的输出差不多是一样的。在此之后是Jar插件的`jar`构建目标的输出，我们显示了这一内容。Jar插件绑定到了Maven构建生命周期的打包阶段，它创建了一个名为`hib-dev-ch12-2.0-SNAPSHOT.jar`的JAR目标artifact。

[1] <http://www.oreilly.com/catalog/9780596517724/>.

使用Maven生成IDE项目文件

在上一节中，你创建了一个数据库、编译了代码、对代码进行了测试并将最终的生成结果打包到JAR中，但是这些还只是Maven能够做到的一部分。Maven有许多插件，不过它们一般不直接附加到Maven的生命周期中。一些项目的一个常见功能就是为集成开发环境（IDE, Integrated Development Environment）提供配置文件。通常一个团队将会使用一种标准的开发环境，如IntelliJ、Eclipse、NetBeans或者Emacs JDE，为这些工具提供的配置文件也将是项目的一部分。如果一个团队具有一套标准的工具，那将IDE配置文件保存在版本控制工具中可能更有意义。

注意：大多数人都有各自喜欢的开发工具，所以，从POM中生成IDE项目可以让每个人都使用他们觉得最有效率的工具。

当团队并没有使用一套统一的工具时，在版本控制工具中保存IDE配置显得没有多大意义。就以大多数开源项目为例，参与项目的每个人之间很少有或根本没有交流，也没有能力协调彼此之间开发平台和IDE的选择。即使你认为不会有项目允许单独的开发人员可以独立地选择自己的开发工具集，从Maven中生成IDE配置文件也有一定的好处。这些配置文件通常都包含了同样的依赖信息、标识符以及设

置，可以将这些信息保存在pom.xml文件中。如果不从pom.xml中生成IDE配置，就必须保存和维护多份这些信息的复本。

为了生成一套Eclipse配置文件，可以从ch12示例目录中运行mvn eclipse: eclipse命令：

```
$mvn eclipse: eclipse
[INFO]Scanning for projects.....
[INFO]Searching repository for plugin with prefix: 'eclipse'.
[INFO]-----
-----
---
[INFO]Building Harnessing Hibernate: Chapter Twelve: Maven
[INFO]task-segment: [eclipse: eclipse]
[INFO]-----
-----
---
[INFO]Preparing eclipse: eclipse
[INFO]No goals needed for project-skipping
[INFO][eclipse: eclipse]
[INFO]Using source status cache: ~\examples\ch12\target\mvn-
eclipse-cache
.properties
[INFO]Wrote settings to~
\examples\ch12\.settings\org.eclipse.jdt.core.prefs
[INFO]Wrote Eclipse project for"hib-dev-ch12"to~\examples\ch12.
```

Maven只是使用pom.xml中的信息来生成项目目录中的Eclipse项目文件（.project和.classpath配置文件）。要将生成的项目导入到Eclipse中，在Eclipse中选择File→Import（导入）菜单选项，在Import对话框中，展开General（普通）部分，选择"Existing Projects into Workspace"（导入现有的项目到工作空间），再点击"Next"按钮。在接下来的界面中通过浏览器导航到示例目录，并选择示例的根目录，

再点击"Next"按钮。Eclipse这时就会在所选择的目录及其所有子目录中搜索.project文件。在成功导入项目以后，还需要添加一个名为M2_REPO的Java Classpath变量，让它指向`~/m2/repository`。为此，打开Eclipse的Preferences设置，在Java/Build Path/Classpath Variables节点下，你可以看到包括ECLIPSE_HOME在内的一系列已经有的Classpath Variables（类路径变量）。添加一个名为M2_REPO的新Classpath Variable，并让它指向你的本地Maven2仓库，在这个例子中应该是`~/m2/repository`。

只需要添加一次M2_REPO类路径变量，而且如果你不喜欢通过Eclipse Preferences来添加类路径变量的方法，也可以使用Maven Eclipse插件将类路径变量添加到你的Eclipse工作空间中，如下所示：

```
/home/tobrien$mvn-o eclipse: add-maven-repo\  
-Declipse.localRepository=~/m2/repository\  
-Declipse.workspace\=~/eclipse
```

本书所有章节的示例都包含了一个pom.xml文件，但它们并非都完全兼容Maven。不过，在每一章中有一件事，你可以放心地去做，那就是使用`mvn eclipse: eclipse`命令来生成Eclipse项目文件。试一下！再换到其他章的示例目录，运行`mvn eclipse: eclipse`，再导入那一章到Eclipse中。或者，如果你想为所有章节一次性地生成Eclipse项目文件，则可以先转到所有章节示例的父目录，再从那里运行`run mvn eclipse: eclipse`命令。从examples目录运行该命令将会为所有子模块执

行Eclipse插件的eclipse构建目标，每一章的代码示例都有一个eclipse构建目标。

除了Eclipse项目，你也可以为IntelliJ的Idea IDE生成项目配置。为此，可以按以下方式执行Idea插件的idea构建目标：

```
$mvn idea: idea
[INFO]Scanning for projects.....
[INFO]Searching repository for plugin with prefix: 'idea'.
[INFO]-----
-----
---
[INFO]Building Harnessing Hibernate: Chapter Twelve: Maven
[INFO]task-segment: [idea: idea]
[INFO]-----
-----
---
[INFO]Preparing idea: idea
[INFO]No goals needed for project-skipping
[INFO][idea: idea]
[INFO]jdkName is not set, using[java version1.6.0_02]as default.
```

idea构建目标会在项目目录中创建3个文件：hib-dev-ch12.iml、hib-dev-ch12.ipr以及hib-dev-ch12.iws，用这些文件就可以将该项目导入到Idea了。

用Maven生成报告

我们已经构建了一个项目，现在准备生成一些简单的报告。一种可能的报告形式就是用于显示单元测试结果的HTML页面。测试成功还是失败了？哪些测试失败了？你也可以为代码提供JavaDoc，对代码添加标注，以方便查阅。所有这些都生成好以后，可以运行命令`mvn site`:

```
$mvn site
[INFO]Scanning for projects.....
[INFO]-----
-----
---
[INFO]Building Harnessing Hibernate: Chapter Twelve: Maven
[INFO]task-segment: [site]
[INFO]-----
-----
---
[INFO]Setting property: classpath.resource.loader.class=
>'org.codehaus
.plexus.velocity.ContextClassLoaderResourceLoader'.
[INFO]Setting property: velocimacro.messages.on=>'false'.
[INFO]Setting property: resource.loader=>'classpath'.
[INFO]Setting property: resource.manager.logwhenfound=>'false'.
[INFO]*****
***
[INFO]Starting Jakarta Velocity v1.4
[INFO][site: site]
Constructing Javadoc information.....
Standard Doclet version 1.6.0_02
Building tree for all the packages and classes.....
Generating~/examples/ch12/target/site/apidocs\index.html.....
[INFO]Generate"Source Xref"report.
[INFO]Generate"Continuous Integration"report.
[INFO]Generate"Dependencies"report.
[INFO]Generate"Issue Tracking"report.
[INFO]Generate"Project License"report.
```

```
[INFO]Generate"Mailing Lists"report.  
[INFO]Generate"About"report.  
[INFO]Generate"Project Summary"report.  
[INFO]Generate"Source Repository"report.  
[INFO]Generate"Project Team"report.  
[INFO]Final Memory: 23M/42M
```

以上输出片段有相当多的删减。如果你运行`mvn site`，将能够看到页面和活动的页面。**Maven**创建了一个项目网站，并生成了一些有用的报告。下面我们快速浏览一下结果。

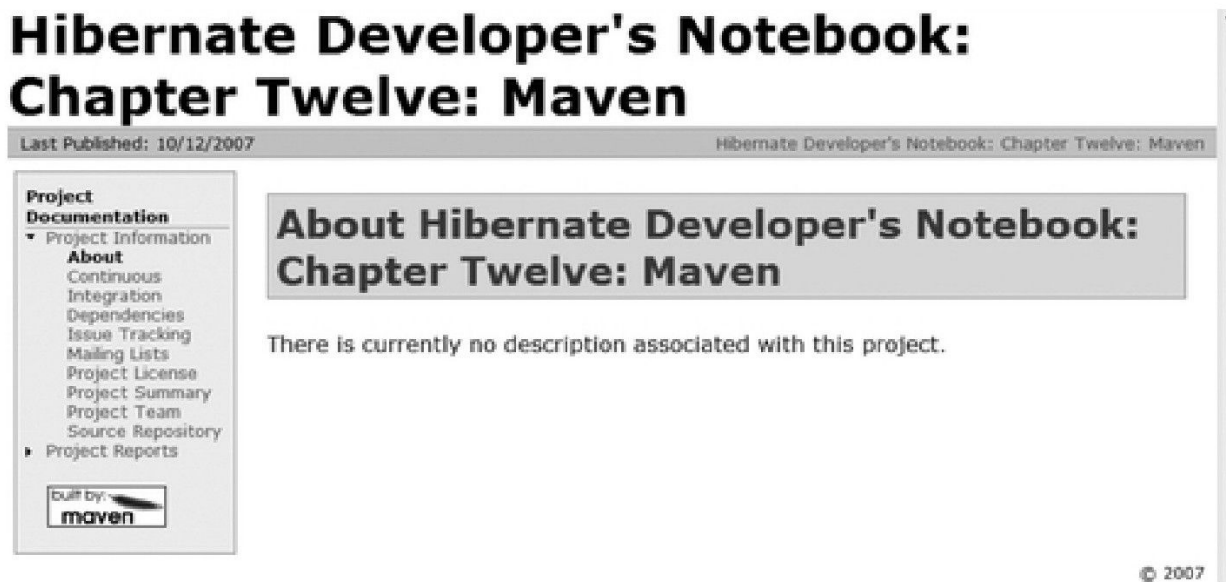


图 12-1 生成Maven的网站

Maven生成的简单网站上列出了许多与项目有关的报告和页面。虽然默认的网站模板看起来不算太好，如图12-1所示，但它确实提供了一个基本的网站，你可以此为基础来在线发布有关项目的信息。如果项目的POM配置正确，就可以生成一个简单的页面，列出了项目的成员、许可协议以及一个指向问题管理工具（如Bugzilla、JIRA或Trac）

的链接。如果你正在创建一个开源项目，这些信息足以构成一个公共的、面向开发人员的网站的坚实基础。即便你是在一个封闭的环境中开发项目，这样的网站对开发团队来说也是有用的。默认Maven网站的感观风格有许多需要改进之处，可以使用保存在src/site目录下的样式表（stylesheets）和模板来进行定制。如果点击导航菜单左边的Project Reports（项目报告）链接，就可以看到一些有用的报告。还可以浏览项目的JavaDoc，如图12-2所示。

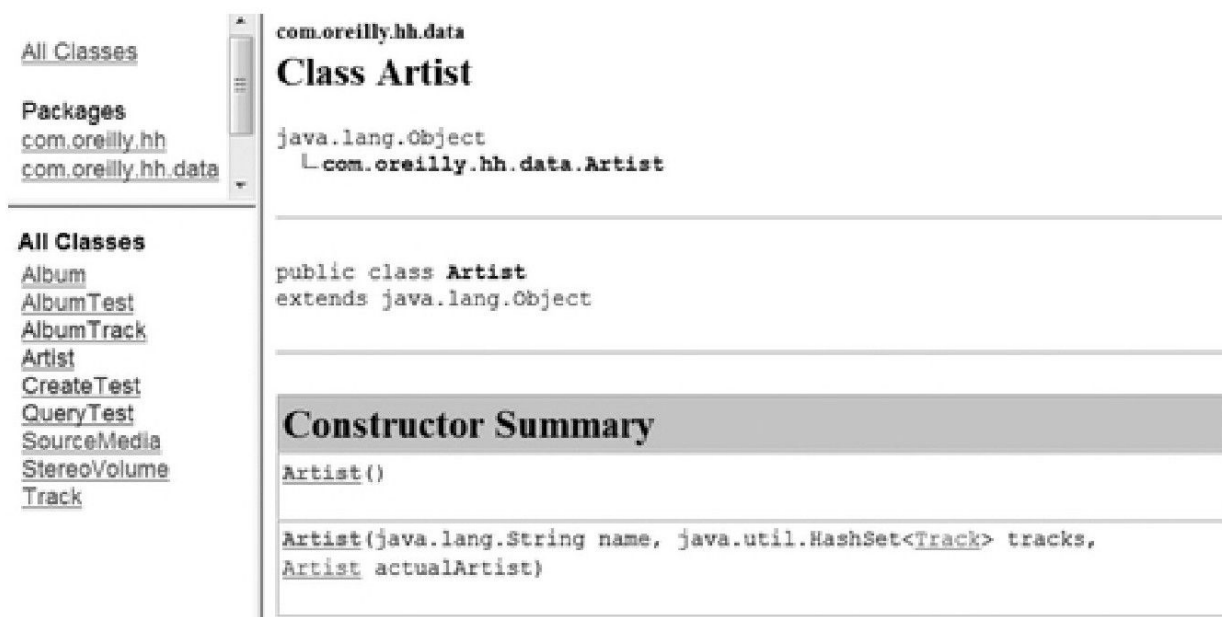


图 12-2 网站内的项目JavaDoc页面

点击Surefire报告链接，就可以浏览单元测试的结果（如图12-3所示）。

Chapter Twelve: Maven

Last Published: 10/13/2007

Hibernate Developer's Notebook: Chapter Twelve: Maven

Project Documentation

- Project Information
- Project Reports
 - JavaDocs
 - Maven Surefire Report**
 - Source Xref
 - Test JavaDocs

built by:


Summary

[Summary][Package List][Test Cases]

Tests	Errors	Failures	Skipped	Success Rate	Time
1	0	0	0	100%	1.275

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Package List

[Summary][Package List][Test Cases]

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
com.oreilly.hh	1	0	0	0	100%	1.275

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

com.oreilly.hh

	Class	Tests	Errors	Failures	Skipped	Success Rate	Time
	ArtistTest	1	0	0	0	100%	1.275

图 12-3 项目的单元测试结果页面

因为这个项目只定义了一个单元测试，所以Surefire报告没有多少内容。如果项目中包含了大量的测试，就可以在这个报告中检查代码的总体质量。如果发现单元测试有问题，或者如果想查找代码中特定的某一错误，就可以使用JXR报告来浏览经过标注的、交叉引用（cross-referenced）的源代码（如图12-4所示）。

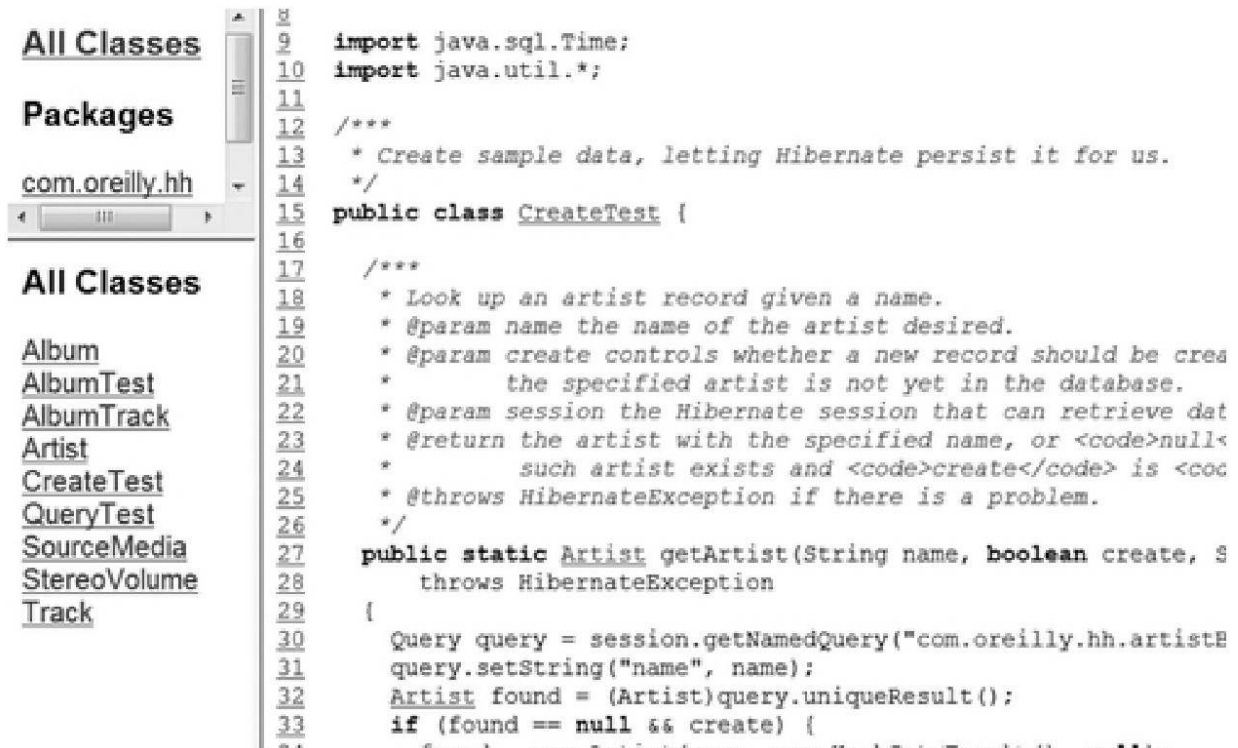


图 12-4 交叉引用的HTML源代码

在查看代码时，JXR报告很有用。还有很多其他报告可以使用，例如，用Clover报告测试的覆盖度，以及JDepend报告、DocBook、PDF生成等。

这里我们不打算详细介绍Maven的工作原理，只是进行了编译、构建数据库、测试、打包、对某个简单的示例代码进行文档化。在下一节中，我们将看看让这一切都成为可能的文件：pom.xml。这样，对Maven的配置和定制，你才会有一定的感觉。

Maven项目对象模型

项目对象模型（Project Object Model）是Maven的核心，它是运行Maven惟一需要的配置文件。每个Maven项目都有一个pom.xml文件，它描述了项目的属性和依赖关系，以及任何自定义的构建配置。例12-3演示了支持前面项目构建的pom.xml文件内容。

例12-3: Maven项目对象模型（POM）

```
<project xmlns="http://maven.apache.org/POM/4.0.0"❶
xmlns: xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi: schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.oreilly.hh</groupId>❷
  <artifactId>hib-dev-ch12</artifactId>
  <version>2.0-SNAPSHOT</version>
  <name>Harnessing Hibernate: Chapter Twelve: Maven</name>
  <packaging>jar</packaging>❸
  <dependencies>❹
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>1.8.0.7</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>3.2.5.ga</version>
      <exclusions>
```

```

<exclusion>
<groupId>javax.transaction</groupId>
<artifactId>jta</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>org.apache.geronimo.specs</groupId>
<artifactId>geronimo-jta_1.1_spec</artifactId>
<version>1.1</version>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.14</version>
</dependency>
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-annotations</artifactId>
<version>3.3.0.ga</version>
</dependency>
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-commons-annotations</artifactId>
<version>3.3.0.ga</version>
</dependency>
</dependencies>
<build>
<extensions>❶
<extension>
<groupId>hsqldb</groupId>
<artifactId>hsqldb</artifactId>
<version>1.8.0.7</version>
</extension>
<extension>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.14</version>
</extension>
</extensions>
<plugins>
<plugin>❷
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<configuration>
<source>1.5</source>
<target>1.5</target>
</configuration>

```

```

</plugin>
<plugin>❶
<groupId>org.codehaus.mojo</groupId>
<artifactId>hibernate3-maven-plugin</artifactId>
<version>2.0</version>
<executions>
<execution>
<id>generate-ddl</id>
<phase>process-classes</phase>
<goals>
<goal>hbm2ddl</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
<reporting>❷
<plugins>
<plugin>
<artifactId>maven-javadoc-plugin</artifactId>
</plugin>
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>jxr-maven-plugin</artifactId>
</plugin>
<plugin>
<artifactId>maven-surefire-report-plugin</artifactId>
</plugin>
</plugins>
</reporting>
</project>

```

这就是我们编译、测试、生成Web网站时需要提供的所有配置：

❶ 文档元素是project。我们采用xsi: schemaLocation来定义Maven POM第4版的XML Schema。如果你想全面地了解引用的这个Schema，可以把它加载到类似XMLSpy的工具中，这个Schema的文档注释一目了然。

❷接着，我们定义了一组每个Maven项目都可以公用的标识符：`groupId`、`artifactId`以及`version`。这三个属性相当于Maven项目的惟一“坐标”。在这个POM的全部内容中，你会注意到各种依赖、扩展以及插件都引用了这三个属性中的一个或多个。

❸这个项目的`packaging`是`jar`。将打包格式设置为`jar`，就告诉Maven这个项目的构建结果应该生成一个JAR文件。Maven支持一套预定的打包格式，例如`pom`、`jar`以及`war`。如果在POM中没有声明`packaging`元素，其默认值就是`jar`。

❹`dependencies`元素告诉Maven这个项目需要依赖什么样的外部库文件。Maven POM中的`dependencies`与我们一直在用的Maven Ant Task中的`dependencies`完全一样。如果将这个`pom.xml`中的`dependencies`的内容与`build.xml`的对应内容进行比较，你会发现这里包括的依赖与第7章的`build.xml`中的依赖是一样的：`JUnit`、`HSQLDB`、`Hibernate`（不带JTA）、`Apache Geronimo`中的JTA以及`Hibernate Annotations`。

❺`build`元素是我们定义构建的地方。在这个地方我们能够为插件配置自定义的行为，以及配置Maven默认安装中可能没有提供的插件。因为准备将`Hibernate3`插件配置为使用`HSQLDB`，所以在`build`元素下的`extensions`添加了`hsqldb`和`log4j`库。`extension`元素的内容是任何在插件执行期间需要在类路径上出现的依赖的`groupId`、`artifactId`以及`version`值。

⑥第一个plugin元素是配置编译器插件，让它使用兼容Java 5的源代码和目标文件。

⑦这里我们对Codehaus ([\[1\]](#)) 的Mojo项目中的Hibernate3 Maven插件进行配置。在编写本书时，Hibernate3插件的最新发行版本是2.0。这一插件配置标明了hibernate.cfg.xml的公共位置，也添加了一个属性来告诉hbm2ddl：当生成数据库模式时不要删除已有的数据库。基于Ant的例子会在编译完成之后的构建期间执行hbm2ddl目标，而POM则是将hbm2ddl放在Maven的process-classes阶段执行。我们将在本章后面的“Maven构建的生命周期”一节中再详细介绍Maven构建生命周期中的各个阶段。

⑧最后，reporting区段配置在网站生成阶段要执行的报告（report）生成器。用于生成报告的Maven插件有一个report目标，在网站生成过程中会调用这个目标。为了包括报告，必须要在reporting元素下包括相应的插件。

注意这种pom.xml文件和其他章节的build.xml文件的区别。在build.xml中，我们需要明确地告诉Ant要做什么，在哪和什么时间进行操作，怎么保存结果。在Maven中，我们只需要指出依赖和说明需要执行的目的。Maven插件知道应该怎么做，负责编译的插件会查找src/main/java目录下的源文件和src/test/java目录下的测试代码。Hibernate3插件“知道”我们的hibernate.cfg.xml文件应该放在

src/main/resources 目录中。如果我们将hibernate.cfg.xml文件保存到不同的位置，就得告诉插件这一情况，但是，为什么要不辞辛苦地修改默认的约定，又有什么意义呢？本章稍后会在12.8节仔细看看Hibernate3插件。

父/子项目对象模型

本章的示例使用的是一个单独pom.xml，以简化对Maven的介绍，而其他各章的示例采用的是所谓的多模块（multi-module）Maven项目，在这种模型中，每一个子模块（submodule）需要依赖一个父模块。我们以第7章为例，在ch07目录中，你可以找到一个pom.xml文件，其内容如例12-4所示。这里需要注意的第一件事就是这个文件这么小，它的所有依赖是在哪定义的？

例12-4：第7章的pom.xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <parent>
    <groupId>com.oreilly.hh</groupId> ❶
    <artifactId>hib-dev-parent</artifactId>
    <version>2.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>hib-dev-ch7</artifactId> ❷
  <packaging>jar</packaging> ❸
  <name>Harnessing Hibernate: Chapter Seven</name>
```

```

<version>2.0-SNAPSHOT</version>
<build>
<plugins>
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>hibernate3-maven-plugin</artifactId>
<version>2.0-alpha-3-SNAPSHOT</version>
<executions>
<execution>
<id>generate-ddl</id>
<phase>process-classes</phase>
<goals>
<goal>hbm2ddl</goal>
</goals>
</execution>
</executions>
<configuration>
<components>
<component>
<name>hbm2ddl</name>❷
<implementation>annotationconfiguration</implementation>
</component>
</components>
</configuration>
</plugin>
</plugins>
</build>
<dependencies>❸
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-annotations</artifactId>
<version>3.3.0.ga</version>
</dependency>
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-commons-annotations</artifactId>
<version>3.3.0.ga</version>
</dependency>
</dependencies>
</project>
</project>

```

这个pom.xml从表面上看起来相当简单，因为它的配置大部分都继承自父项目的POM（稍后再介绍）。我们先来看看这个简单的POM

的内容:

❶我们先把这个项目链接到一个父项目上，由`groupId`、`artifactId`以及`version`这三个属性来定义该父项目。在Maven中定义的所有项目都具有这三个属性，它们是一个Maven项目的惟一“坐标”。父项目将在下一个例子中再介绍，不过，现在你只需要知道`examples/ch07/pom.xml`继承了`examples/pom.xml`中定义的所有属性就可以了。

❷我们将`artifactId`属性定义为`hib-dev-ch07`。此处改写了父POM中的相应属性，为这个项目提供了一个惟一的值。如前所述，所有项目必须具有一个惟一的由`groupId`、`artifactId`以及`version`构成的组合，这样可以确保例ch07具有一个惟一的`artifactId`。注意，这个`pom.xml`并没有定义`groupId`，因为它会继承父`pom.xml`中的`groupId`。

❸接下来，我们改写了这个项目的POM的`packaging`属性。Maven有一套预定义的打包格式，例如`pom`、`jar`以及`war`。将例ch07的打包格式修改为`jar`，就告诉Maven：这个项目期望它的构建结果是生成JAR文件。

❹这一行是在配置Maven Hibernate3插件，类似于我们在本章示例目录中进行过的处理。这里不同的是，我们正在配置Hibernate3插件，让它使用基于标注的配置方式，第7章介绍的全部是有关标注的内容。

这一配置将会从一套保存在src/main/java中的带有标注的模型类中抽取Hibernate的映射信息。Maven先将Java源文件编译到target/classes目录中，Maven Hibernate3插件再扫描生成的字节码文件，查找其中的@Entity标注。最后，hbm2ddl目标再用这些映射信息生成HSQLDB数据库。

❶我们声明这个子模块的依赖为Hibernate Annotations和Hibernate Commons Annotations。这些新的依赖会添加到父项目定义的任何依赖当中。

第7章的POM引用了一个父POM，那这个父POM是在哪定义的？转到ch07的父目录，你会看到也有一个pom.xml文件（文件体积要大很多）。在这个文件中就定义了供其他各章示例的POM所共享的所有属性。我们来看看这个父pom.xml，其内容如例12-5所示。

例12-5：共享的顶级pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.oreilly.hh</groupId>❶
  <artifactId>hib-dev-parent</artifactId>
  <packaging>pom</packaging>❷
  <name>Harnessing Hibernate: Parent Project</name>
  <version>2.0-SNAPSHOT</version>
  <modules>❸
  <module>ch01</module>
```

```
<module>ch02</module>
<module>ch03</module>
<module>ch04</module>
<module>ch05</module>
<module>ch06</module>
<module>ch07</module>
<module>ch08</module>
<module>ch09</module>
<module>ch10</module>
<module>ch11</module>
<module>ch12</module>
<module>ch13</module>
<module>ch14</module>
</modules>
<build>
<extensions>❹
<extension>
<groupId>hsqldb</groupId>
<artifactId>hsqldb</artifactId>
<version>1.8.0.7</version>
</extension>
<extension>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.14</version>
</extension>
</extensions>
<resources>
<resource>
<directory>src</directory>❺
</resource>
</resources>
<sourceDirectory>src</sourceDirectory>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>❻
<configuration>
<source>1.5</source>
<target>1.5</target>
</configuration>
</plugin>
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>hibernate3-maven-plugin</artifactId>❼
<version>2.0</version>
<configuration>
<components>
```

```
<component>
<name>hbm2java</name>
<implementation>configuration</implementation>
</component>
</components>
<componentProperties>
<drop>false</drop>
<configurationfile>
src/hibernate.cfg.xml
</configurationfile>
<jdk5>true</jdk5>
</componentProperties>
</configuration>
</plugin>
</plugins>
</build>
<dependencies> ⑧
<dependency>
<groupId>hsqldb</groupId>
<artifactId>hsqldb</artifactId>
<version>1.8.0.7</version>
</dependency>
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate</artifactId>
<version>3.2.5.ga</version>
<exclusions>
<exclusion>
<groupId>javax.transaction</groupId>
<artifactId>jta</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>org.apache.geronimo.specs</groupId>
<artifactId>geronimo-jta_1.1_spec</artifactId>
<version>1.1</version>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.14</version>
</dependency>
</dependencies>
</project>
```

该父POM包含的信息要比其子POM多很多。它定义了所有公共的依赖、目录设置以及每章示例可以共享的插件配置。再一次，我们一部分一部分地看看这个父POM：

❶ `groupId`元素用于定义供所有子项目共享的`group`标识符。
`groupId`的标准命名习惯是使用一个反向的完全限定的域名（你自己的域名，或是与项目相关的域名），其后再跟上特定领域的标识符。如果你想在Maven仓库中发布自己的`artifact`，这样的命名规则就特别重要。

❷ 在父POM中，将`packaging`设置为`pom`。这意味着这个项目的存在只是为了提供POM，这个项目不会提供任何JAR文件或WAR文件，它只提供POM。

❸ 父POM定义了`modules`列表。这部分列出了每章的示例目录，每个目录中都包含一个`pom.xml`文件。如果你查看这些目录，就会发现其中的每个`pom.xml`文件都引用了这个父POM。在父POM中指定所有的模块，就可以让我们能够一次性地构建所有章节（或模块）的项目。

❹ `extensions`元素用于配置各插件使用的类路径。父`pom.xml`定义的`extensions`与例12-3中的相同。

❸此处，我们正在定制项目的类路径资源（resource）的位置。在通常的Maven项目中，我们不需要定义这个属性，而是依赖其默认值`${basedir}/src/main/resources`（`${basedir}`是包含pom.xml文件的目录）。在本书中（因为这是本关于Hibernate的书，而不是专门介绍Maven的），我们要将Maven构建应用到现有的章节示例上，所以需要将类路径资源定义为`src/`。除了定义类路径资源的位置，我们也需要定义源代码的位置。在标准的Maven项目中，我们应该不用在pom.xml中定义这个属性，而是使用默认值`${basedir}/src/main/java`。

❹这一部分用于将编译器插件配置为Java 5目标。

❺Hibernate3插件是通过几个全局选项来配置的。将`drop`设置为`false`，会让`hbm2ddl`目标不删除已有的数据库。将`jdk5`设置为`true`，会让`hbm2java`目标生成支持Java 5泛型的模型类属性（例如使用`Set<Album>`，而不只是`Set`）。我们也可以用`configurationFile`来配置`hibernate.cfg.xml`的默认位置（如果该文件不在Maven的预想位置上）。`hbm2java`的`component`项中，将`implementation`设置为`configuration`，这意味着默认行为将是扫描主资源目录，以查找`.hbm.xml`映射文件。我们在例12-4中看到过，`ch07`的pom.xml改写了`hbm2ddl`组件的这一配置，使其采用标注来作为Hibernate的配置方式。

❻最后，顶级的POM定义了一组供所有章节的例子共享的依赖。它们是HSQLDB 1.8.0.7、Hibernate 3.2.5.ga、Apache Geronimo项目的

JTA库以及Log4j 1.2.14。可以比较一下这个dependencies元素的内容与任何一章的build.xml脚本中Maven Ant Task的配置。

以运行ch07目录中的示例来做为例子。首先，删除data目录（如果存在），再运行mvn install。这将编译源代码、按模型中的标注来生成数据库模式、将ch07中的示例打包到一个JAR文件中。在运行构建过程以后，就可以像原来用Ant那样来运行CreateTest、QueryTest以及AlbumTest。这些过程如例12-6所示。

例12-6：使用Maven来运行第7章的测试

```
~/examples/ch07$mvn exec: java-
Dexec.mainClass=com.oreilly.hh.CreateTest
[INFO]Scanning for projects.....
[INFO]Searching repository for plugin with prefix: 'exec'.
[INFO]-----
-----
---
[INFO]Building Harnessing Hibernate: Chapter Seven
[INFO]task-segment: [exec: java]
[INFO]-----
-----
---
[INFO]Preparing exec: java
[INFO]No goals needed for project-skipping
[INFO][exec: java]
~/examples/ch07$mvn exec: java-
Dexec.mainClass=com.oreilly.hh.QueryTest
[INFO]Scanning for projects.....
[INFO]Searching repository for plugin with prefix: 'exec'.
[INFO]-----
-----
---
[INFO]Building Harnessing Hibernate: Chapter Seven
[INFO]task-segment: [exec: java]
[INFO]-----
-----
```

```
---
[INFO]Preparing exec: java
[INFO]No goals needed for project-skipping
[INFO][exec: java]
Track: "Russian Trance", 00: 03: 30
Track: "Video Killed the Radio Star", 00: 03: 49
Track: "Test Tone 1", 00: 00: 10
~/examples/ch07$mvn exec: java-
Dexec.mainClass=com.oreilly.hh.AlbumTest
(output omitted)
```

以上我们使用了一个通常不附加到Maven生命周期中的插件构建目标—`exec: java`。这个目标将接受从命令行传递的`exec.mainClass`参数，并执行给定类中的`main ()`方法。

[1] <http://mojo.codehaus.org/maven-hibernate3/hibernate3-maven-plugin/>.

Maven构建的生命周期

在本章的前面几节中我们谈到了生命周期，所以到目前为止，你应该至少掌握Maven的构建生命周期中包含编译、测试、打包、安装阶段。在例12-3中，我们将hbm2ddl目标附加到类处理（process-classes）阶段，同时指出这个阶段紧跟在编译阶段之后。如果所有这些看起来让你感到一头雾水，不要担心，我们将会简单地解释生命周期中的各个阶段和它们的执行顺序。当从命令行运行Maven时，需要选择指定某个阶段的名称，或者显式地执行一个插件目标。当指定了一个阶段时，Maven将执行它的生命周期中直到该阶段之前的所有阶段（包括这个指定的阶段）。在Maven处理其构建生命周期时，它会触发附加到当前阶段上的插件的目标。另一方面，如果显式地请求一个插件目标，则Maven将只执行那个单独的插件目标。

以下列举了Maven的所有生命周期阶段，而且按它们发生的顺序进行排序。对于我们的示例来说比较重要的阶段，会解释得细致些；而对于其他不重要的阶段，则将它们组合起来以节省空间：

validate（验证）

验证POM。

generate-sources（生成源代码）

生成任何源代码。Hibernate3插件有一个hbm2java目标，可以根据基于XML的映射而生成Java源代码（就像我们在前几章中做的那样）。如果我们想为映射生成Java源代码，则可以将hbm2java附加到这个阶段。

process-sources、**generate-resources**、**process-resources**（处理源文件、生成资源、处理资源）

除了生成源代码，还需要处理源代码生成的输出、生成资源文件（例如properties文件、图片、声音以及程序包的其他非代码元素）以及处理资源文件。

compile（编译）

编译器插件运行编译目标，对编译源代码根目录下的所有源代码进行编译。插件可以在编译源代码根目录中添加新的目录，例如Hibernate3插件的hbm2java目标就会根据.hbm.xml文件生成源代码，并将生成的源代码放到target/generated-source目录中。Hibernate3插件接着再将target/generated-source目录添加到编译源代码根目录，以便在编译时可以包含新生成的文件。

process-classes（后处理编译生成的类）

该阶段对编译的结果进行后期处理（`post-processes`）。我们已经将`hbm2ddl`目标挂到了类的后期处理任务上，因为需要对`hbm2ddl`任务进行配置，才可以利用对象模型中的标注，将它们编译为模型类文件。`Hibernate3`插件被配置为需要对生成的字节码进行搜索，以查找出现的"`@Entity`"标注。因为只有先编译好了模型类，才可以进行这样的扫描，所以我们将这个目标挂到了编译阶段之后的处理阶段。

`generate-test-sources`、`process-test-sources`、`generate-test-resources`、`process-test-resources`

这几个阶段与从`generate-sources`到`generate-resources`的几个阶段类似，可以生成单元测试及其需要的任何资源。

`test-compile`（编译测试源码至测试目标目录）

使用`testCompile`目标，再次调用`Compiler`插件来编译测试源代码。对于我们的示例来说，这个它只是编译`src/test/java`目录中的一个简单的类。`testCompile`目标会编译测试源代码根目录中的所有源代码。再说一次，这并不是简单地对`src/test/java`中的所有代码进行编译，因为在前面4个生命周期阶段中，插件都有机会生成单元测试的源代码（在测试源代码根目录中添加目录）。

`test`（测试）

Surefire插件扫描test-compile阶段生成的继承自JUnit的TestCase的类，并使用JUnit执行这些单元测试（也可以运行TestNG测试）。

prepare-package（准备打包）

在真正将项目打包以前，执行一些准备处理。提供与此相关的目标。

package（打包）

对于我们的示例项目来说，最终只需要生成一个JAR文件。jar打包的默认行为就是创建一个名为\${artifactId}-\${version}.jar的文件。如果需要的话，可以自定义这个文件名称。不过，如果可能的话，尽可能避免对Maven进行定制，将是更好的做法。

pre-integration-test、integration-test、post-integration-test（预集成测试、集成测试、集成测试后期）

单元测试通常不需要连接到数据库，事实上，在单元测试中连接数据库的做法也让人感觉有些古怪（我将其称为功能或集成测试）。如果你需要执行一系列集成测试，可以使用这3个阶段来定义相关的目标。

verify（验证）

可以为验证阶段定义一些用于检查生成的输出是否符合质量规范的目标。

install（安装）

在我们的项目中，安装阶段只是将生成的artifact复制到`~/.m2/repository/${groupId}/${artifactId}/${version}/${artifactId}-${version}.jar`文件中。如果你正在创建一套复杂的交叉依赖的项目（也就是说，项目A定义了对一个具体的库的依赖，而项目B也需要依赖这个库），那么项目B就可以用`groupId`、`artifactId`以及`version`（版本）来定义一个对项目A的依赖，当Maven在构建项目B时，会尝试在本地仓库中解析项目A的artifact。通过将制品也安装到本地仓库中，`install`阶段就可以支持这一功能。

deploy（部署）

用于将项目的制品、网站、报告发布到一个外部的仓库中。项目也可以利用这个阶段将各种制品发布到一个远程Maven仓库。远程Maven仓库可以是公共的或私有的Maven仓库，或者是企业内部的Maven仓库。

使用Maven Hibernate3插件

本章使用的是来自Mojo项目中的Maven Hibernate3插件（[\[1\]](#)），Mojo是Codehaus上的一个项目，它为开发Maven插件提供一个Apache Software Foundation（ASF）以外的空间。不像ASF那样有很多规则，只要对一些Maven插件感兴趣就可以开始为Mojo做贡献，这比ASF上Maven项目中的那些插件的开发要更容易些。将Mojo放在Codehaus上的另一个原因是，一些Maven插件使用的技术的许可协议与Apache Software License不兼容。Mojo的插件可以采用GPL协议，而且可以按非Apache Software License发布。而在Apache Software Foundation中，所有东西的发行都需要采用Apache Software License。

为了使用Maven Hibernate3插件，所有你需要做的就是将例12-7演示的plug-in元素包括到项目的pom.xml文件中。

例12-7：使用Maven Hibernate3插件

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns: xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi: schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
.....pom content skipped.....
<build>
<plugins>
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>hibernate3-maven-plugin</artifactId>
<version>2.0</version>
```



```
</plugin>
</plugins>
</build>
.....pom content skipped.....
</project>
```

Hibernate3插件定义了以下构建目标（也称为配置目的的组件）：

hibernate3: hbm2cfgxml

基于现有的数据库模式而生成一个**hibernate.cfg.xml**文件。可以使用这个目标为已有的数据库构建相应的**Hibernate**配置。

hibernate3:hbm2ddl

基于现有的**Hibernate**映射而生成相应的**SQL DDL**（数据库定义语言）（使用标注或**.hbm.xml**文件）。在我们的例子中，它生成的是一个**HSQLDB**数据库。

hibernate3:hbm2doc

生成一个**HTML**报告，用于描述**Object Model**（对象模型）、数据库模式以及如何将二者关联起来。生成的报告的界面风格与**JavaDoc**类似。

hibernate3:hbm2hbmxml

根据现有的数据库模式而生成.hbm.xml映射文档。如果你有一套数据库表，就可以用这个任务来生成映射文件，再用下一个构建目标来生成Java类。

hibernate3:hbm2java

根据Hibernate映射而生成Java源代码。这个构建目标可以从.hbm.xml文件生成Java代码，或者生成与现有数据库中的数据表对应的Java类。

有关Maven插件的更多信息，可以参阅Maven网站（[\[2\]](#)）的在线文档；有关Hibernate3 Maven插件的信息，还可以看看Mojo项目（[\[3\]](#)）。

除了Apache Maven发布的核心插件以外，人们还需要为Maven编写自己的插件，Mojo这个项目就是为此服务的。如果你有了关于Maven插件的好想法，应该考虑参与到Mojo项目之中。

配置Hibernate3插件

Maven Hibernate3插件提供了许多配置点。以下列举了其中一些重要的配置选项：

propertyfile

如果在Hibernate配置文件中没有指定JDBC连接信息，也可以在database.properties文件中指定JDBC连接参数，默认使用src/main/resources/database.properties。

configurationfile

Hibernate配置文件。默认使用src/main/resources/hibernate.cfg.xml。

jdk5

生成Java 5源代码。当生成与Hibernate映射匹配的Java源代码时，支持使用泛型和枚举类型。默认值是false。

ejb3

生成带有ejb3标注的源代码，默认值是false。

drop

在执行hbm2ddl前，先删除数据库表。默认值是false。

create

在执行hbm2ddl时创建一个数据库。默认值是true。

outputfilename

为Hibernate3插件的构建目标配置输出文件名。

`implementation`

配置Hibernate映射的源文件。有效值可以是`configuration`、`annotationconfiguration`、`jpaconfiguration`以及`jdbcconfiguration`。

还有其他几个配置参数，如果需要完整的列表，可以查阅Maven Hibernate3项目网站上的Component Properties ([\[4\]](#)) 页面。

注意：Hibernate3插件目前还在由Mojo进行开发当中，不要忘记经常查看项目网站上的更新信息。

可以为某个组件（`hbm2java`或`hbm2ddl`）配置它的每个配置点，也可以为所有组件配置它们的每个配置点。由于例12-3并没有提供任何配置属性，所以Hibernate会按照默认配置从`.hbm.xml`文件中读取Hibernate映射。在例12-4中，`plug-in`配置改写了`hbm2ddl`目标的`implementation`设置，让Hibernate从类中的标注来读取映射信息。这一点值得再详细讨论一下。

属性`implementation`有点特殊，它可以设置Hibernate3插件获得Hibernate映射信息的方式。有4种选择：选择`configuration`，将在源文件目录中搜索`.hbm.xml`映射文件；选择`annotationconfiguration`，将在编译过的类中搜索Hibernate Annotations；`jdbcconfiguration`利用反向工程

(reverse engineering)，直接从现有的数据库中生成Hibernate映射；jpaconfiguration与annotationconfiguration类似，只是它查找的是persistence.xml，而不是hibernate.cfg.xml配置文件。按照构建目标运行的环境，每个目标都有其默认的implementation设置。例如，在JDK 1.4运行环境下，hbm2ddl目标的默认值是configuration；而在Java SE 5运行环境下，它的默认值就是annotationconfiguration。有关所有Hibernate3构建目标的implementation默认值，可以参阅Maven Hibernate3项目网站上的Component Configuration ([\[5\]](#)) (组件配置) 页面。

生成Hibernate映射文档

Maven Hibernate3插件有一个任务是可以为一套Hibernate映射创建文档。该文档与JavaDoc类似，详细描述了数据库表的结构和相应的Java模型对象。为了生成这个文档，需要在运行install以后再运行Hibernate3插件的hbm2doc构建目标：

```
~/examples/ch12$mvn install
(output omitted)
~/examples/ch12$mvn hibernate: hbm2doc
[INFO]Scanning for projects.....
[INFO]Searching repository for plugin with prefix: 'hibernate3'.
[INFO]-----
-----
[INFO]Building Harnessing Hibernate: Chapter Twelve: Maven
[INFO]task-segment: [hibernate3: hbm2doc]
```

```
[INFO]-----  
-----  
---  
[INFO][hibernate3: hbm2doc]  
[INFO]using annotationconfiguration task.  
[INFO]Configuration XML file loaded:  
file: /Users/tobrien/svnw/hibernate-  
book/current/examples/ch12/src/main/resources/hibernate.cfg.xml  
[INFO]src/main/resources/database.properties not found within the  
project.  
Trying absolute path.  
[INFO]No hibernate properties file loaded.  
[INFO]-----  
-----  
[INFO]BUILD SUCCESSFUL  
[INFO]-----  
-----
```

构建目标hbm2doc执行完成以后，将会生成HTML文档
target/hibernate3/javadoc/index.html。在浏览器中打开这个页面，就会看到一组描述Hibernate映射细节的HTML文档。可以从数据库表或实体的视图来浏览Hibernate映射。从实体视图（如图12-5所示）可以查看是哪个数据库表关联到了某个特定实体对象，而数据库表视图（如图12-6所示）则是先点击一个数据库表，再看看是哪个实体对象映射到了这个特定的表。

[All Entities](#)
Packages

All Entities
[Album](#)
[AlbumTrack](#)
[Artist](#)
[StereoVolume](#)
[Track](#)

[Tables](#) [Entities](#)

com.oreilly.hh.data

Entity Track

com.oreilly.hh.data.Track

Identifier Summary

Name	Column	Type	Description
<u>id</u>	Column	Integer	

Property Summary

Name	Column	Access	Type	Description
<u>added</u>	<u>added</u>	field (get / set)	Date	
<u>artists</u>		field (get / set)	Set<Artist>	
<u>comments</u>		field (get / set)	Set<String>	
<u>filePath</u>	<u>filePath</u>	field (get / set)	String	
<u>playTime</u>	<u>playTime</u>	field (get / set)	Date	
<u>sourceMedia</u>	<u>sourceMedia</u>	field (get / set)	SourceMedia	
<u>title</u>	<u>TITLE</u>	field (get / set)	String	
<u>volume</u>	<u>left</u> <u>right</u>	field (get / set)	<u>StereoVolume</u>	

图 12-5 hbm2doc生成的Track实体文档

超越Maven

Maven一定可以为你节省不少时间。我敢肯定，一定是这样的。如果把**Maven**用得恰到好处，你将不必再为维护一套本来已经足够聪明的程序式构建系统而花费精力（参阅：过度工程（**over-engineered**））。但是，和其他任何技术一样，**Maven**自身也存在一些问题：它的开发文档仍然有些粗糙，甚至有些插件（例如**Hibernate3**插件）还没有完整的文档。

注意：如果**Mojo**提供的**Hibernate3**插件文档不能让你感到满意，应该批评**Tim**了。他现在正和**Mojo**的维护人员一起努力改进这个插件。希望当你读到这本书时，**Hibernate3**插件已经具备完整的文档了。如果还没有，那就随便给他发个邮件催一下吧。

向开发团队中引入某种技术非常不容易，尤其是那些对构建结构或数据库设计之类的东西有特殊要求的技术。我曾经见到过有些团队拒绝**Hibernate**，是因为**Hibernate**的合集映射不符合数据库管理员的标准。也见到过某些组织拒绝**Hibernate**，是因为它创建的SQL达不到开发团队认可的标准。对**Hibernate**的批评还有些是非常抽象的，听起来好像是什么时髦的艺术评论家的言论（“关系数据库是一种过时的结构”）。对于类似的这些不理性的批评，**Maven**也不是它们的陌生人。

无论如何，我们当中肯定就有这样的人，因为可能必须接受某种标准，所以就不愿意使用可以节省时间的技术。如果你决定采用Maven这种构建工具，那么在开始使用它之前，务必要保证团队中的每个人都要接受这种声明式（declarative）构建的思想。

在下一章中，我打算介绍另一种通过提供标准和约定来节省更多时间的技术。你刚看到Maven是如何负责项目的构建，才让你有时间坐下来放松一会儿。第13章将向你演示如何用Spring Framework来完成Hibernate的大部分编码工作。就像Maven让项目构建变得不费力气一样，Spring让使用Hibernate变得更加容易。使用这种节省时间的技术也会有一定的风险：只需要花费片刻的延迟就完成了很多繁杂的任务，你的老板可能因此发现原来你的工作这么简单，所以就开始给你分配一堆堆的任务。如果你担心像管理构建过程或编写模板代码这样繁杂的活都做完了，我建议你不必在意本章和下一章介绍的内容。另一方面，你可以接受Maven和Spring，而用节省下来的时间多休息一会儿，早点回家，或者在Safari上阅读更多的在线电子图书也未尝不可。

第13章 Spring入门: Hibernate与Spring

Spring是什么

如果你关注一下最近几年Java编程的发展，或许应该听说过Spring Framework。Spring Framework一个反转控制（Inversion of Control, IoC）容器，并集成了许多附加功能和模块。在某些方面，它确实只是一个IoC容器；而在另一些方面，它也是一个完整的应用程序开发平台（我们将在下一节再具体介绍这些概念的含义）。

Spring最初由Rod Johnson在2000年创建，现在它已经发展成为有望代替Enterprise Java Beans（EJB）的成熟框架。在2000年到2004年之间，Sun Microsystems发布了一系列针对EJB的规范。原来如果不编写大量重复性代码，并且/或使用专有的工具来屏蔽底层技术，那么将很难理解、开发以及部署EJB。Rod创造性地编著了《Expert One-on-One J2EE Design and Development》（2002年由WROX出版），这本书向广大开发人员介绍了如何用简单的IoC容器和一系列封装API来取代EJB，而这些API可以将你的程序与Sun的底层API隔离开来。在Spring中不需要直接与JDBC、JMS或JTA API直接交互，而是应该使用Spring提供的各种模板类和辅助类。虽然Sun的核心企业API目前仍然很重要，但使用Spring的一个作用就是放宽了Sun在定义或提议的新库和应

用上的限制。Spring不是直接对JDBC或JNDI进行编程，而是在Spring容器内编写自己的代码进行操作，这样就不必了解实现持久化、消息传递以及其他处理时所使用的底层技术。作为打破Sun在Java界定制标准的垄断地位的一种尝试，Spring并不是现在惟一的项目，但它无疑是最成功的。

你可能会写一些自己的持久化层对象来直接与JDCB交互，但是也可以使用Oracle某种定制的专有代码来实现一部分，而用Hibernate实现剩余的其他部分。Spring通过提供一个神秘的IoC容器和一组有用的抽象，就可以支持这种选择。有关企业开发的讨论不再只是涉及Sun和Java；Community Process（JCP）定制的各种标准；相反，Spring通过提供一种公用的“总线”（bus），在此基础上可以构建大多数新的项目应用，从而让开发企业应用有了更多选择。以Spring与Web框架的集成为例：Wicket、Struts 2（原来的WebWork）、Stripes、GWT以及Tapestry都提供了与Spring的优秀集成，而Spring的文档也详细介绍了与JavaServer Faces、Tapestry以及WebWork的集成。Spring也为各种对象关系映射（Object Relational Mapping）框架提供了方便的集成，包括Hibernate、JDO、Oracle TopLink、iBatis SQL Maps以及JPA。Spring作为一个平台，可以让开发人员“自由选择”各种实现，在一定程度上来说，现在几乎所有新的开源库或项目都必须支持Spring Framework，才能让更多的开发人员愿意使用它。业内许多人士一致认为，虽然Sun可能会声称Java是一个“平台”，但它其实只是一种编程语言；而

Spring才是一种真正的平台。当你读完本章以后，就会对Spring如何接管Hibernate的大量工作有所理解。

什么是反转控制

对于IoC还没有一个明确的定义，但比较集中的一个就是依赖注入（Dependency Injection）。在Wikipedia上有以下定义：

它是指一种模式，其中，对象创建和链接的控制由对象转移到了工厂。

作为一种轻量级容器，Spring负责将一套组件集成在一起，通过JavaBean属性或构造参数注入依赖对象的实例。使用Spring，你可以先开发一系列简单的对象，再告诉Spring怎么把它们串接起来以创建一个更大的系统。如果你的系统架构依赖的是可重用的“组件”，这样的实现方法就非常方便。在本章中，我们就用Spring Application Context将数据访问对象（Data Access Objects, DAO）实现为这种组件（或bean），而示例类则通过bean的属性来依赖这些组件。如果想在其他系统中（例如命令行工具或Web应用程序）重用这些DAO类，那么系统可以按bean属性的形式来依赖同样的组件，Spring会以一种描述程序结构的XML文档为基础，把各种组件都串接起来。

Spring负责将一套功能集中的组件串接在一起，通过这种方法来促进组件的重用。你的DAO不必关心它们连接到什么组件，或是什么组件会使用它们；它们只是为依赖于它们的组件提供了一定的接口。如果你对依赖注入或反转控制的更多细节感兴趣，可以阅读一下Martin Fowler的《Inversion of Control Containers》和《Dependency Injection Pattern》（[\[1\]](#)）。

组合Spring和Hibernate

现在你应该知道Spring是做什么用的了，我们回头再看看Hibernate。本章集中介绍DAO模式、事务抽象以及Spring Framework提供的工具类。

你希望能从本章学到什么？虽然通过Spring来使用Hibernate相当直接，不过在我们可以真正利用Spring的Hibernate集成之前，还需要几步准备工作。首先，因为Spring是一个IoC容器，所以需要修改前几章的示例，以“串接”的方式来创建对象。我们在下一节才会研究DAO模式。在创建好DAO以后，将修改目前的这套示例，实现一个公共接口Test，并为它应用一个Transactional标注。接着，为了创建Spring应用程序上下文，我们要编写一个XML文档，告诉Spring要创建什么对象以及和什么对象串接起来。最后，需要写一个命令行应用程序，用来加载Spring应用程序上下文和启动示例程序。

注意：介绍足够了，接下来看看代码吧。

添加Spring框架为项目依赖

为了在示例项目中使用Spring，需要为构建过程再添加一个依赖。打开build.xml，添加例13-1中用粗体突出显示的依赖。

例13-1：添加Spring依赖

```
<artifact: dependencies pathId="dependency.class.path">
  <dependency
groupId="hsqldb"artifactId="hsqldb"version="1.8.0.7"/>
  <dependency
groupId="org.hibernate"artifactId="hibernate"version="3.2.4.sp1">
    <exclusion groupId="javax.transaction"artifactId="jta"/>
  </dependency>
  <dependency groupId="org.hibernate"artifactId="hibernate-
tools"version=
"3.2.0.beta9a"/>
  <dependency groupId="org.hibernate"artifactId="hibernate-
annotations"
version="3.3.0.ga"/>
  <dependency groupId="org.hibernate"artifactId="hibernate-
commons-annotations"
version="3.3.0.ga"/>
  <dependency
groupId="org.apache.geronimo.specs"artifactId="geronimo-
jta_1.1_spec"version="1.1"/>
  <dependency groupId="log4j"artifactId="log4j"version="1.2.14"/>
  <dependency
groupId="org.springframework"artifactId="spring"version="2.5"/>
</artifact: dependencies>
```

很好，现运行Ant build，就会看到Maven Ant task将从某个Maven仓库中下载spring-2.5.jar。

[1] <http://martinfowler.com/articles/injection.html>.

编写数据访问对象

数据访问对象（Data Access Object, DAO）是一种常见的设计模式，用于分离应用程序业务逻辑代码和访问，处理数据库记录的代码。在更大型的体系结构中，DAO层可以作为两个独立的体系层次的边界。我们以Spring Framework为背景来介绍DAO对象，以便让你对如何将Hibernate应用到整体系统架构中有所了解；同时，之所以要介绍DAO对象，也是因为在现实世界需要与数据库进行交互的系统中，DAO对象代表了一种常见的模式。

什么是数据访问对象

如果有作者用20多页的篇幅来介绍DAO模式、它的优点和缺点，不免会让我感到吃惊。如果你看过Sun J2EE蓝图，它就用很多篇幅解释了如何使用工厂方法的DAO创建模式、如何在更大的方法中使用DAO模式以支持企业应用程序的开发。我们准备省略很多拘泥于形式的细节，而将DAO模式简单地概括为两点。DAO模式如下：

- 将所有持久化操作（创建、读取、更新、删除）归结到一个接口中，通常按数据表或对象类型进行组织。

·该接口具有多种可切换的实现，可以支持任意各类的持久化API和底层存储介质。

或者，一个更精简的定义是“DAO将持久化的所有繁琐细节隐藏在接口之后”。

当使用像Hibernate这样的对象/关系映射（Object/Relational Mapping, ORM）框架时，通常是将数据库作为一套对象来处理。本书的示例涉及3个对象：Artist、Album、Track，与这些对象相应，我们打算创建3个DAO对象：ArtistDAO、AlbumDAO、TrackDAO。图13-1简单地演示了要创建的ArtistDAO的类图。

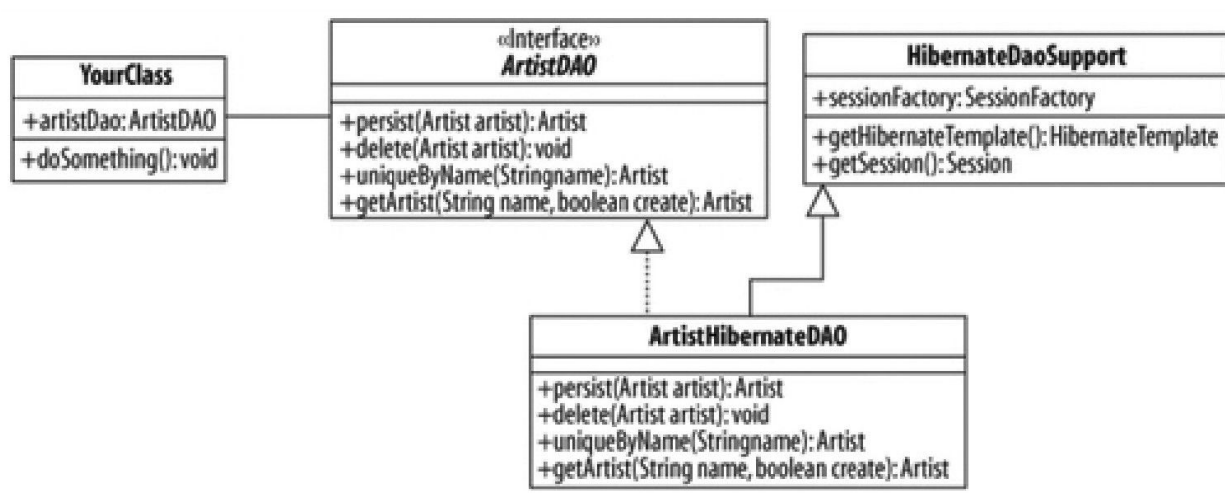


图 13-1 用数据访问对象来分离应用的业务逻辑和持久化代码

在这个图中，你的应用程序的业务逻辑表示为YourClass类，这个类有一个对ArtistDAO接口实例的引用，该接口定义了4个简单的方法：

Artist persist (Artist artist)

将一个**Artist**对象的状态保存到数据库中。根据**artist**参数的状态，这个方法将插入一个新记录行，或更新一个已有的记录行。这个方法的约定是检查**artist**参数的**id**属性：如果**id**属性是**null**，就向**ARTIST**表插入一个新行；如果**id**属性不为**null**，就更新数据库中的相应行。该方法返回一个持久化的**Artist**对象，换句话说，如果你为这个方法传递一个**id**属性为**null**的**Artist**对象，它会在数据库中创建一行新的记录，并返回一个**id**属性不为**null**的**Artist**对象，这时的**id**属性包含的就是新插入记录的标识符。

void delete (Artist artist)

从数据库中删除匹配的记录。

Artist uniqueByName (String name)

返回姓名等于**name**参数值的一个**Artist**对象。

Artist getArtist (String name, boolean create)

查找姓名匹配**name**参数值的**Artist**对象。如果**create**参数为**true**，则当没有找到匹配的**Artist**对象时，该方法就用提供的**name**参数来创建并持久化一个新的**Artist**对象。

虽然在YourClass类的代码中引用的是ArtistDAO接口，但实际上调用的是ArtistDAO接口的一个实现—ArtistHibernateDAO。该ArtistDAO的实现类继承了Spring的HibernateDaoSupport类，这个Spring提供的工具类包含了所有必需的魔力，让Hibernate代码的编写变得尽可能简单。使用HibernateDaoSupport，你可以不必编写前面几章中看到的所有异常处理、事务管理以及session管理代码。事实上，我想你一定感到惊讶（也可能有些失望），在Spring中使用Hibernate变得这么简单。

本书使用以下命名约定，DAO接口在com.oreilly.hh.dao包中进行定义，它的类名由相关联的对象名称再附加上"DAO"组成（例如ArtistDAO）。将该接口的Hibernate特定实现命名为ArtistHibernateDAO，放在com.oreilly.hh.dao.hibernate包中。

为什么要使用DAO

之所以要使用DAO有很多原因，但第一个也是最重要的原因是这种模式可以增加灵活性。因为是对接口进行编码，当使用了不同的O/R映射服务或存储介质时，就可以方便地构建新的实现。在前面的介绍中曾经提过，Spring提供的集成层可以使用许多对象-关系映射技术，从iBatis SQL Maps到Oracle的TopLink，再到Hibernate，但Spring也提供了一套丰富的抽象，让用JDBC执行SQL变得相当直接。在我经历过的大多数应用程序开发中，Hibernate提供了大部分持久化逻辑，但它并

不能解决所有问题。有时当需要直接执行SQL时，直接使用Spring提供的JdbcTemplate这些类就非常方便。如果在应用程序的业务逻辑层和持久层之间插入DAO接口，这样在需要的时候，就可以更容易地切换到其他特定DAO类或方法的实现。这种灵活性和分离性的优点体现在两个方面，其一是可以容易地替换特定DAO类的实现，其二是当需要重写或更新应用程序的业务逻辑时，可以重用已有的持久化层。本书的许多读者现在遇到的情形是：开发团队一直在维护一个用Struts 1.x编写的遗留系统，而你想将应用程序升级到Stripes或Struts 2。如果持久化代码紧密地耦合到了Web框架代码中，你可能会发现如果不重写其中的一方面，就不可能重写另一方面。

对于这么简单的一个应用程序，采用DAO模式似乎显得没有必要，但是更经常遇到的情况是，你需要在多个项目中重用持久化代码。DAO对象看起来似乎与敏捷开发相背离，但这种在复杂度上的付出会随着时间的推移而体现出其价值。如果你的系统不是那种简单的"Hello World"例子，就可能需要花些时间将持久化逻辑从应用程序的业务逻辑中分离出来。

编写ArtistDAO接口

不耽误时间了，我们看看这个示例的代码编写。需要做的第一件事就是编写ArtistDAO接口。在com.oreilly.hh.dao包中创建一个名为

ArtistDAO的接口，并将例13-2所示的代码放到这个新接口中。

例13-2: ArtistDAO接口

```
package com.oreilly.hh.dao;
import com.oreilly.hh.data.Artist;
/**
 *Provides persistence operations for the Artist object
 */
public interface ArtistDAO{
/**
 *Persist an Artist instance (create or update)
 *depending on the value of the id
 */
public Artist persist (Artist artist) ;
/**
 *Remove an Artist from the database
 */
public void delete (Artist artist) ;
/**
 *Return an Artist that matches the name argument
 */
public Artist uniqueByName (String name) ;
/**
 *Returns the matching Artist object.If the
 *create parameter is true, this method will
 *insert a new Artist and return the newly created
 *Artist object.
 */
public Artist getArtist (String name, boolean create) ;
}
```

好，这段代码看起来相对比较容易，这里我们只要介绍几个简单方法的实现。接下来就看看如何使用HibernateDaoSupport类来实现其处理逻辑。

实现ArtistDAO接口

下一步就应该编写ArtistDAO的实现。我们打算编写一个ArtistHibernateDAO类来实现ArtistDAO接口，并继承Spring的HibernateDaoSupport类。HibernateDaoSupport提供了对Hibernate Session对象和HibernateTemplate的访问，可以用HibernateTemplate来简化对Hibernate Session对象任意的操作。为了实现ArtistDAO，先在com.oreilly.hh.dao.hibernate包中创建一个新类，让它包含例13-3所示的ArtistDAO接口的特定于Hibernate的实现。

例13-3: 实现ArtistDAO接口

```
package com.oreilly.hh.dao.hibernate;
import java.util.HashSet;
import org.apache.log4j.Logger;
import org.hibernate.Query;
import
org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import com.oreilly.hh.dao.ArtistDAO;
import com.oreilly.hh.data.Artist;
import com.oreilly.hh.data.Track;
/**
 *Hibernate-specific implementation of the ArtistDAO
interface.This class
 *extends the Spring-specific HibernateDaoSupport to provide
access to
 *the SessionFactory and the HibernateTemplate.
 */
public class ArtistHibernateDAO extends HibernateDaoSupport
implements ArtistDAO{
private static Logger log=
Logger.getLogger (ArtistHibernateDAO.class) ;
/* (non-Javadoc)
 *@see com.oreilly.hh.dao.ArtistDAO#persist
(com.oreilly.hh.data.Artist)
 */
public Artist persist (Artist artist) {❶
return (Artist) getHibernateTemplate () .merge (artist) ;
}
```

```

    /* (non-Javadoc)
    * @see com.oreilly.hh.dao.ArtistDAO#delete
    (com.oreilly.hh.data.Artist)
    */
    public void delete (Artist artist) {❷
        getHibernateTemplate ().delete (artist) ;
    }
    /* (non-Javadoc)
    * @see com.oreilly.hh.dao.ArtistDAO#uniqueByName
    (java.lang.String)
    */
    public Artist uniqueByName (final String name) {❸
        return (Artist) getHibernateTemplate ().execute (new
HibernateCallback () {
        public Object doInHibernate (Session session) {
            Query query=getSession ().getNamedQuery
("com.oreilly.hh.artistByName") ;
            query.setString ("name", name) ;
            return (Artist) query.uniqueResult () ;
        }
    }) ;
    }
    /* (non-Javadoc)
    * @see com.oreilly.hh.dao.ArtistDAO#getArtist (java.lang.String,
boolean)
    */
    public Artist getArtist (String name, boolean create) {❹
        Artist found=uniqueByName (name) ;
        if (found==null&&create) {
            found=new Artist (name, new HashSet<Track> () , null) ;
            found=persist (found) ;
        }
        if (found!=null&&found.getActualArtist () !=null) {
            return found.getActualArtist () ;
        }
        return found;
    }
}

```

❶这个persist () 方法只是简单地调用HibernateTemplate的merge () 方法。merge () 方法的实现会检查artist参数的id属性。如果id为null, merge () 方法就向ARTIST表插入一行新的记录, 并返回一个带

有生成的id属性的Artist新实例。如果id属性不为null, merge () 方法就会先查找匹配的记录行, 再用artist参数的内容来更新这一记录。

❷delete () 只是将artist参数传递给HibernateTemplate的delete () 方法。该方法需要接收一个其id属性不为null的对象, 并删除ARTIST表中的相应数据行。

❸uniqueByName () 才是更有趣的开始。此处是我们在这个类中第一次引用Session对象, 在整本书中曾经一直在使用它。首先用getSession () 获取一个NamedQuery。接着再设置命名参数name, 取回一个惟一的查询结果。如果数据库中没有匹配的Artist, uniqueResult () 将返回null。相信你也注意到这里使用了一个匿名的HibernateCallback实例, 并将它传递给HibernateTemplate对象。有关HibernateCallback类的更多信息, 可以参阅本章后面的13.2.6节。

❹getArtist () 方法实际上只是将查询转交给了ArtistDAO中的其他方法。它通过调用uniqueByName (), 按照名称来取回一个Artist对象。如果没有找到任何Artist对象, 而且create参数为true, getArtist () 方法就会创建一个id属性为null的Artist实例, 再调用persist () 方法。如果没有找到匹配的Artist对象, 而且create参数为false, 这个方法就会返回null。如果找到的或新创建的Artist对象具有一个非null的actualArtist属性, 这个方法将返回artist.getActualArtist () 的值。(这一步骤的目的可以参阅第5.5节的解释。)

HibernateDaoSupport的功能

HibernateDaoSupport可以让我们连接到SessionFactory，但不必知道Hibernate环境是怎么创建或配置的。当将HibernateDaoSupport子类化（subclass）到我们的DAO类后，就可以通过getSession（）方法访问Hibernate Session，通过getHibernateTemplate（）方法访问HibernateTemplate。你应该已经知道用Hibernate Session对象可以做什么（就是本书前面10章的内容）。Spring/Hibernate集成中有趣的部分是由HibernateTemplate类提供的，我们接下来就深入研究一下这个类的细节。

根据HibernateTemplate的Javadoc中的说明：“可以用这个类来取代对原始Hibernate 3 Session API的使用。”HibernateTemplate简化了原本用Session对象完成的任务，同时将Hibernate-Exception转换为多个通用的DataAccessException。可以用两种方法来使用HibernateTemplate：调用一套简单的工具函数，例如load（）、save（）、delete（）；或者使用execute（）方法来执行一个HibernateCallback实例的调用。在大多数情况下，使用HibernateTemplate的简单工具函数就足够了；只有要在HibernateTemplate内执行某些Hibernate特定的代码时，才需要创建一个HibernateCallback对象。

DataAccessException是什么？在前面介绍DAO设计模式时，就说过它的目的就是要向应用程序代码屏蔽属于任何持久化API或库的特殊

性。如果我们独立于特定技术的DAO抛出一个Hibernate特定的ObjectNotFoundException异常，就对我们没什么帮助了。所以Hibernate-Template就需要负责处理可能在其中发生的任何Hibernate特定的异常。Stripes API通过将这些特定实现的异常封装到它自己通用的数据访问异常中，提供了一种对这种异常进行包容的简单方法。

HibernateTemplate和HibernateCallback是帮助我们避免编写一行又一行不必要的Java代码的实际骨干。接下来就使用它们两个来重新实现前面几章中的示例。

应该怎么做

在使用HibernateTemplate和HibernateCallback之前，我们需要先大致看看它们提供的方法。HibernateTemplate提供了很多简单方便的方法，可以将原来直接使用Hibernate Session API时需要多行代码才能完成的功能压缩到只需要简单的一行。我们以查询数据库表为例，来看看这种简单方法到底是怎么回事。例13-4演示了查询和搜索对象的几个示例。

例13-4: HibernateTemplate的find () 工具方法

```
HibernateTemplate tmpl=getHibernateTemplate ();
//All of these lines Find Artist with name'Pavement'
List artists=tmpl.find ("from com.oreilly.hh.data.Artist a"+
"where a.name='Pavement'"); ❶
String name="Pavement";
```

```

List artists=tmp1.find ("from com.oreilly.hh.data.Artist a"+
    "where a.name=?", name) ; ❷
List artists=tmp1.findByNameParam ("from
com.oreilly.hh.data.Artist a"+
    "where a.name=: name", "name", name) ; ❸
//Assuming that there is a NamedQuery annotation"Artist.byName"on
the
//Artist class
List artists=tmp1.findByNameQuery ("Artist.byName", name) ; ❹
Artist artist=new Artist () ;
artist.setName ("Pavement") ;
List artists=tmp1.findByExample (artist) ; ❺
//If we want to iterate through the result
Iterator artists=tmp1.iterate ("from com.oreilly.hh.data.Artist"+
    "where a.name=?", name) ; ❻
//The following lines find all Artists
List artists=tmp1.find ("from com.oreilly.hh.data.Artist") ; ❼
List artists=tmp1.loadAll (Artist.class) ;

```

Find方法相对比较直接:

❶这是一个简单的find () 方法, 接受一个不带参数的HQL查询语句。

❷该方法的这个版本接受一个HQL查询语句, 以及一个附加的参数。类似地另一个版本将接受一个查询语句和一组附加的参数: List find (String hql, Object[]params)。这些方法都支持非命名查询参数的使用, 但正如第3章所述, 编写查询还有更好的方法。

❸findByNameParameter () 方法可以处理带有命名参数的查询。

❹findByNameQuery () 方法可以让你快速调用一个预定义的HQL查询, 在这个例子中是名为"Artist.byName"的查询。

⑤通过`findByExample ()`方法，还可以使用Hibernate的示例查询（`query-by-example`）的功能。

⑥如果想循环遍历查询结果，可以调用`iterate ()`方法。当调用`iterate ()`方法时，Hibernate首先取回匹配数据行的所有ID，当通过返回的`Iterator`遍历结果时，再初始化各个元素。

⑦最后，如果只是想加载给定某个表的所有数据行，则可以调用`find ()`或`loadAll ()`方法。

正如第3.4节所讨论的，命名查询是一种将查询定义移出DAO代码的好办法。如果你使用的是标注，可以使用`@NamedQuery`标注来定义命名查询。有关该标注的更多细节可以参阅第7章的相关内容。

如果我们已经知道了某个持久对象的ID值，就可以用`HibernateTemplate`提供的工具方法通过ID来加载对象，如例13-5所示。

例13-5：用`HibernateTemplate`加载对象

```
//Identifier of Artist to load
Integer id=1;
//Load an Artist object, return persistent Artist object
Artist artist=getHibernateTemplate ().load (Artist.class, id) ;
//Populate the object passed in as a parameter.Using the
//object's type to specify the class
Artist artist=new Artist () ;
getHibernateTemplate ().load (artist, id) ;
```

在第1个例子中，我们用一个Class和序列化ID值来调用load（）函数。Hibernate将从数据库中检索对应的记录，并返回请求对象的实例。除了使用Class对象，你也可以传递一个对象实例，Hibernate会用参数的类型来决定检索哪个类。

我们已经看过了HibernateTemplate中用于查询和加载对象的工具方法。那么怎样修改数据库中的记录呢？例13-6演示了几个在数据库中插入和更新记录的示例。

例13-6：用HibernateTemplate保存和更新记录

```
//Persist a new instance in the database
Artist a=new Artist ();
a.setName ("Fischerspooner");
getHibernateTemplate ().save (a);
//Load, modify, update a row in the database
Artist a=getHibernateTemplate ().load (Artist.class, 1);
a.setName ("Milli Vanilli");
getHibernateTemplate ().update (a);
//Either insert or update depending on the identifier
//of the object; associate resulting object with Session
Artist a=getHibernateTemplate ().merge (a);
```

Save（）和update（）方法也比较直观，这两个方法与Hibernate Session对象上的同名方法类似。Save（）方法生成一个新的ID，并向数据表中插入一行新的记录；update（）方法则更新数据库表中的匹配记录。merge（）方法更灵活些：它会检查参数的id属性，按照ID是否为null来调用save（）或update（）方法。

也可以通过HibernateCallback，使用Hibernate Session来执行任何SQL语句。在详细解释这一用法之前，我们先看看例13-7。

例13-7：编写HibernateCallback

```
final String name="Pavement";
Artist artist= (Artist) getHibernateTemplate () .execute (new
HibernateCallback () {
    public Object doInHibernate (Session session) {
        Criteria criteria=session.createCriteria (Artist.class) ;
        criteria.add (Restrictions.like ("name", name) ) ;
        return criteria.uniqueResult () ;
    }
}) ;
```

那么这个例子中到底发生了什么？这个示例首先实例化一个实现了HibernateCallback接口的匿名内部类，并将它传递给HibernateTemplate的execute () 方法。HibernateCallback接口只定义了一个方法doInHibernate ()，需要向它传递一个Hibernate Session。在这个方法的内部（在我们的匿名内部类中实现的）使用Hibernate Criteria API生成查询，按姓名检索回一个Artist对象。

为什么当我们本来能够容易地获得Hibernate Session的引用，而且也能创建同样的Criteria对象时，却要使用回调方法？即使在HibernateDaoSupport中使用getSession () 方法能够直接访问Session对象，我们还是希望避免直接调用Hibernate API，因为我们不想抛出任何Hibernate特定的异常（甚至也不想抛出RuntimeException）。需要记住，你的应用程序正在通过一个接口来访问这个DAO，它不知道也不

关心Hibernate特定的ObjectNotFoundException或HQL中的异常。不是直接用getSession () 访问Session对象，你能够也应该向你的应用程序屏蔽这些底层处理细节，这就是为什么要在HibernateTemplate中用HibernateCallback来运行Hibernate API调用的原因了。

其他DAO在哪里

因为这些DAO处理都非常相似，所以不值得在这里一一列出和讨论其他DAO处理了。你可能不想手工输入所有代码，如果还想看看它们的话，只要下载代码示例就可以了。

创建应用程序上下文对象

前面介绍Spring时，我们讨论了它如何负责创建和连接应用程序中的组件。为了让Spring组装各组件，我们需要告诉它系统中有什么组件（Spring将它们称为bean）以及它们之间是如何连接在一起的。我们使用一个XML文档来描述每个bean的类，为每个bean分配一个ID，建立起各个bean之间的关联关系。为什么要用ID？在Spring中，ID就是每个bean的惟一的逻辑名称，在表达bean之间的关系，以及在运行时请求bean时，就会用到bean的ID属性。在我们的示例Spring配置文件中，就使用了诸如artistDao和albumDao之类的逻辑名称，每个ID引用了在配置文件中定义的一个组件。

Spring再用这个XML文档来创建一个ApplicationContext对象，利用这个对象我们就可以按名称取回相应的组件。图13-2是我们这个程序ApplicationContext的内容结构图表。

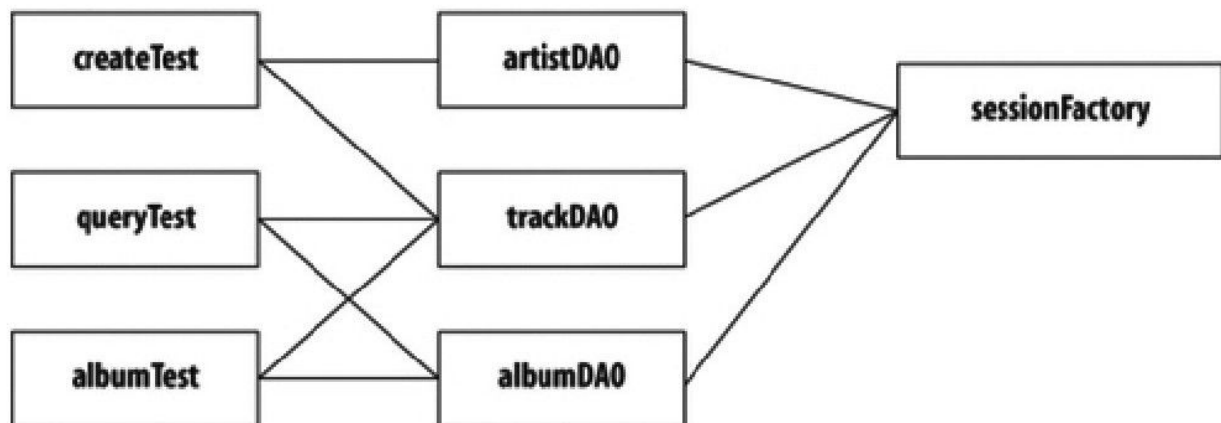


图 13-2 我们的Spring应用程序上下文

从图13-2中可以看到，我们的3个测试组件与3个DAO对象连接在一起，每个DAO对象都有一个会话工厂对象引用，它负责创建一个Hibernate会话对象和连接到数据库。这个应用程序的Spring配置文件如例13-8所示，应该将它命名为applicationContext.xml，并放在src目录下。

例13-8: Spring的配置文件applicationContext.xml

```
<?xml version="1.0"encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation=
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd"
default-lazy-init="true">❶
<bean id="sessionFactory"❷
class="org.springframework.orm.hibernate3.annotation.Annotation
SessionFactoryBean">
<property name="annotatedClasses">❸
<list>
<value>com.oreilly.hh.data.Album</value>
<value>com.oreilly.hh.data.AlbumTrack</value>
<value>com.oreilly.hh.data.Artist</value>
<value>com.oreilly.hh.data.StereoVolume</value>
<value>com.oreilly.hh.data.Track</value>
</list>
</property>
<property name="hibernateProperties">❹
<props>
<prop key="hibernate.show_sql">false</prop>
<prop key="hibernate.format_sql">true</prop>
<prop key="hibernate.transaction.factory_class">org.hibernate.
transaction.JDBCTransactionFactory</prop>
```

```

    <prop key="hibernate.dialect">org.hibernate.dialect.HSQLDialect
    </prop>
    <prop key="hibernate.connection.pool_size">0</prop>
    <prop key="hibernate.connection.driver_class">org.hsqldb
    .jdbcDriver</prop>
    <prop key="hibernate.connection.url">jdbc: hsqldb: data/music;
    shutdown=true</prop>
    <prop key="hibernate.connection.username">sa</prop>
    <prop key="hibernate.connection.password"></prop>
    </props>
  </property>
</bean>
<!--enable the configuration of transactional behavior based on
annotations-->
<tx: annotation-driven transaction-manager="transactionManager"/
> ⑤
  <bean id="transactionManager"
  class="org.springframework.orm.hibernate3.HibernateTransactionMan
  ager">
    <property name="sessionFactory">
      <ref local="sessionFactory"/>
    </property>
  </bean>
  <bean
  class="org.springframework.beans.factory.annotation.RequiredAnnotati
  on
  BeanPostProcessor"/> ⑥
  <!--Define our Data Access beans-->
  <bean
  id="albumDAO"class="com.oreilly.hh.dao.hibernate.AlbumHibernateDAO"
  > ⑦
    <property name="sessionFactory"ref="sessionFactory"/>
  </bean>
  <bean
  id="artistDAO"class="com.oreilly.hh.dao.hibernate.ArtistHibernateDAO
  ">
    <property name="sessionFactory"ref="sessionFactory"/>
  </bean>
  <bean
  id="trackDAO"class="com.oreilly.hh.dao.hibernate.TrackHibernateDAO"
  >
    <property name="sessionFactory"ref="sessionFactory"/>
  </bean>
  <!--Define our Test beans-->
  <bean id="createTest"class="com.oreilly.hh.CreateTest"> ⑧
    <property name="trackDAO"ref="trackDAO"/>
    <property name="artistDAO"ref="artistDAO"/>
  </bean>

```

```
<bean id="queryTest"class="com.oreilly.hh.QueryTest">
<property name="trackDAO"ref="trackDAO"/>
</bean>
<bean id="albumTest"class="com.oreilly.hh.AlbumTest">
<property name="albumDAO"ref="albumDAO"/>
<property name="artistDAO"ref="artistDAO"/>
<property name="trackDAO"ref="trackDAO"/>
</bean>
</beans>
```

嗯，要读的XML代码量还不小，不是吗？其实在这个文件中还有很多有趣的东西，接下来就细细梳理一下这个文件中的每个部分：

❶ 顶级元素是`<beans>`，为了让Spring能够正常运行，我们必须为其声明一些重要的命名空间。

<http://www.springframework.org/schema/beans>命名空间是用于描述声明的bean元素的默认命名空间，而

<http://www.springframework.org/schema/tx>命名空间则用于定义标注驱动的事务配置，本章稍后会介绍这一内容。`<default-lazy-init>`属性控制Spring IoC容器的默认行为。如果该默认设置为`true`，则只有当请求组件时，Spring才会实例化这些组件。如果把`default-lazy-init`设置为`false`，则Spring在ApplicationContext初始化期间就实例化相关的bean。

❷ 会话工厂这个bean负责生成会话对象，处理到JDBC DataSource的连接。通常，会话工厂将使用一个DataSource，我们在applicationContext.xml中可以为会话工厂配置一个Commons DBCP或

C3P0连接。为了保持这个例子的完整性，会话工厂的配置直接包含了用于配置JDBC连接的各属性。

❸和在hibernate.cfg.xml文件中进行配置一样，此处是在定义需要Hibernate处理的标注类。

❹hibernateProperties元素用于配置Hibernate的设置。稍后在本章的13.3.1节再深入介绍这一配置的细节。

❺事务管理相关的标注配置稍后在本章的13.4.1节再深入介绍。
tx: annotation-driven元素和transactionManager定义可以让我们使用Transactional标注来定义应用程序中任何事务的范围和属性。

❻RequiredAnnotationBeanPostProcessor是一个没有命名的组件，有了这个组件，就可以为带有Required标的setter方法激活一些强制处理。如果将这个Required标注属性放在必需的bean属性的setter方法上，Spring就会在初始化一个bean后，再验证这个属性是否被设置。在测试类中可以用这种方法来确保Spring已经配置好了我们的DAO依赖。

❼在这里定义好了DAO对象：albumDAO、artistDAO以及trackDAO。

❽在这里定义好了test bean：createTest、queryTest以及albumTest。

Hibernate配置属性

仔细看看例13-8中的hibernateProperties部分，其中有许多有趣的配置属性，分别解释如下：

hibernate.connection.driver_class 、 hibernate.connection.url 、
hibernate.connection.username 、 hibernate.connection.password

这几个配置属性负责配置数据库的JDBC连接。这些属性与前面几章中的配置属性类似，它们在applicationContext.xml中的属性值与早先在hibernate.cfg.xml和hibernate.properties中的取值完全一样。

hibernate.connection.pool_size

这个属性用于设置Hibernate内部连接池（connection pool）容量的大小。如果不使用Hibernate提供的连接池，也可以使用Hibernate内建的支持Apache Commons DBCP或C3P0，对于产品级系统的部署，这两种连接池都是很好的选择。如果将这个属性值设置成一个非零的数值，Hibernate就会负责回收和重用数据库的连接。

这种情况很有趣，因为我想为这个示例关掉连接池，以简化HSQLDB的使用，所以将pool_size设置为0。当最后一个数据库连接终止时，HSQLDB需要接收一个SHUTDOWN命令，因为在这里我不想再编写和配置一个专门的关闭脚本，而是简单地保证在用完JDBC Connection对象后就马上关闭它。

`hibernate.dialect`

这个属性用于设置Hibernate方言（`dialect`）。现在Hibernate支持的所有方言列表，请参阅附录C。

`hibernate.transaction.factory_class`

在这个示例中，我们使用JDBC驱动程序来管理数据库事务。更复杂的部署环境需要使用JTA，如果我们使用容器托管的事务，则可以将该属性配置为`org.hibernate.transaction.JTATransactionFactory`。

`hibernate.show_sql`、`hibernate.format_sql`

如果将`show_sql`设置为`true`，Hibernate将打印输出它正在执行的SQL语句。如果你在调试Spring，想弄清楚某个映射是怎么访问数据库表的，这个配置属性就很有用。如果将`format_sql`设置为`true`，就会格式化输出的SQL语句；如果将`format_sql`设置为`false`，SQL语句只打印输出到一行。

把所有组件装配在一起

如果我们不知道如何创建一个Spring ApplicationContext和运行我们的代码，所有这些Spring配置也就没什么用途。在本节，我们打算调整前面的示例CreateTest、QueryTest、AlbumTest类，实现一个Test接口，不是直接从命令行运行它们，而是创建一个TestRunner来执行这些从Spring ApplicationContext获取的测试对象。

Transactions: 测试接口

本章稍后会编写一个名为TestRunner的类，这个类会知道怎么从Spring ApplicationContext中取回一个bean，这个bean应该实现Test接口，需要调用执行该bean的run（）方法。TestRunner使用的bean源于对前面几章的CreateTest、QueryTest以及AlbumTest的修改调整。为了支持这种新的运行方式，我们让它们分别实现一个公共的Test接口，如例13-9所示。

例13-9: Test接口

```
package com.oreilly.hh;
import
org.springframework.transaction.annotation.Transactional;
/**
 *A common interface for our example classes.We'll need this
 *because TestHarness needs to cast CreateTest, QueryTest, or
 *AlbumTest to a common interface after it retrieves the bean
```



```
*from the Spring application context.  
*/  
public interface Test{  
/**  
*Runs a simple example  
*/  
@Transactional (readOnly=false)  
public void run () ;  
}
```

这个Test接口用于为TestRunner提供一个公共接口，它也为我们添加Transactional标注提供了一种方便的方法。Transactional标注负责将一个Session对象绑定到当前线程，启动一个事务处理，如果方法正常返回，则提交事务，如果有异常抛出，则回滚事务。

有关@Transactional标注的更多信息，请参阅附录D。

如何激活事务标注

为了打开Transactional标注的处理，需要在我们的applicationContext.xml文件中添加以下一段配置信息：

```
<!--enable the configuration of transactional behavior based on  
annotations-->  
<tx: annotation-driven transaction-manager="transactionManager"/>  
<bean id="transactionManager"  
class="org.springframework.orm.hibernate3.HibernateTransactionMa  
nager">  
<property name="sessionFactory">  
<ref local="sessionFactory"/>  
</property>  
</bean>
```

tx: annotation-driven元素简单地激活了Transactional标注，将它指向一个PlatformTransaction-Manager。HibernateTransactionManager是Spring Framework的PlatformTransactionManager接口的一个实现。它负责将来自会话工厂的一个Hibernate Session对象用会话工厂Utils绑定到当前线程（Thread）。因为我们的DAO对象都继承自HibernateDaoSupport，也都使用HibernateTemplate，所以这些持久化对象就能够参与到事务管理当中，并获得相同线程上的会话对象。这不仅是因为事务处理的需要，在使用延迟加载的关联时，它也是必需的。Transactional标注可以确保在执行标注过的方法时，让同一会话对象保持打开，并绑定到当前线程。如果没有这个标注，Hibernate就会为每个需要会话的操作都创建一个新的会话对象，你也就不能取回原来用Hibernate检索到的对象上的任何关联。

为什么会这样？让我们回顾一下第5章介绍过的主题。在Hibernate 3中，映射对象之间的关联默认都使用延迟加载。除非为某个类或关联显式地改变这种默认加载方式，直到你访问某个特定对象时，才真正从数据库中检索相关联的对象。例如，如果从数据库中检索回一个Album对象，直到你调用album.getAlbumTracks（）方法时，才会从数据库中再检索回AlbumTrack的List列表对象。为此，Hibernate要做两件事：

1.Hibernate返回一个“代理”对象，用于代表还没有加载的对象。当检索Track对象时，返回的对象是一个Track，不过相关联的集合（例如track.getArtists（））则是PersistentSet的一个实例。

2.PersistentSet由Hibernate负责管理，你通常不需要考虑这一对象。与这里的讨论相关的是，它是PersistentCollection的一个实现，包含了对会话对象的一个引用。换句话说，在按照需要而获取相关联的Artists时，涉及的是PersistentSet。你可以取回一个Track对象，但是在调用track.getArtists（）之前，并不会取回任何Artist对象；而且即便取回关联的Artist对象，也得再次通过会话对象。

只有当PersistentSet引用了一个活动的会话对象时，延迟加载关联才有效果。如果没有一个打开的会话，这时再试图访问延迟加载的关联时，就会抛出一个异常。在Web应用程序中，可以使用Spring的OpenSessionInViewFilter之类的东西来确保在一个请求中持有对一个会话对象的引用。在这个应用程序中，我们依靠Transactional标注来确保run（）方法实现的所有代码都可以访问到同一个Hibernate会话对象。

调整CreateTest、QueryTest以及AlbumTest

现在我们已经定义好了Test接口，而且为这个接口的实现还建立了一个稳定的事务管理环境。接下来就可以修订我们原来的

CreateTest、QueryTest以及AlbumTest类。首先按例13-10所示来修改CreateTest类。

例13-10：为了在Spring中使用而修改CreateTest类

```
package com.oreilly.hh;
import java.sql.Time;
import java.util.*;
import com.oreilly.hh.dao.*;
import com.oreilly.hh.data.*;
/**
 *Create sample data, letting Hibernate persist it for us.
 */
public class CreateTest implements Test{
    private ArtistDAO artistDAO;
    private TrackDAO trackDAO;
    /**
     *Utility method to associate an artist with a track
     */
    private static void addTrackArtist (Track track, Artist artist) {
        track.getArtists ().add (artist) ;
    }
    /* (non-Javadoc)
     *@see com.oreilly.hh.Test#run ()
     */
    public void run () {
        StereoVolume fullVolume=new StereoVolume () ;
        Track track=new Track ("Russian
Trance", "vol2/album610/track02.mp3",
        Time.valueOf ("00: 03: 30") , new HashSet<Artist> () , new Date
        () ,
        fullVolume, SourceMedia.CD, new HashSet<String> () ) ;
        addTrackArtist (track, artistDAO.getArtist ("PPK", true) ) ;
        trackDAO.persist (track) ;
    }
    public ArtistDAO getArtistDAO () {return artistDAO; }
    public void setArtistDAO (ArtistDAO artistDAO) {
        this.artistDAO=artistDAO;
    }
    public TrackDAO getTrackDAO () {return trackDAO; }
    public void setTrackDAO (TrackDAO trackDAO) {
        this.trackDAO=trackDAO;
    }
}
```

}

注意CreateTest类有两个私有的成员变量：artistDAO和trackDAO，它们都配备了作为bean属性的访问器（accessor）方法。接着，按照Test接口的规定，我们实现了一个简单的run（）方法，它最终会调用trackDAO.makePersistent（）来完成Track对象的持久化。所有的处理就是这样的，没有try/catch/finally块，也没有涉及事务管理。在DAO类的帮助下，我们差不多将所有持久化处理都交给了Spring框架。例13-11是从applicationContext.xml摘取的一段代码，该配置将CreateTest类创建成一个ID为createTest的bean，并将它的artistDAO和trackDAO属性组装为相应DAO bean的引用。

例13-11：配置createTest bean

```
<bean id="createTest" class="com.oreilly.hh.CreateTest">
  <property name="trackDAO" ref="trackDAO"/>
  <property name="artistDAO" ref="artistDAO"/>
</bean>
```

将CreateTest的这个实现与例3-3的原始版本进行比较，你会发现它现在已经面目全非了。非Spring版本的CreateTest类必须负责维护会话对象的创建、事务管理、异常处理以及配置。而最新的版本甚至连会话的影子也没有看到。事实上，在CreateTest的最新版本中没有一点Hibernate特定的东西：DAO类让我们的应用程序的业务逻辑不必直接处理底层的持久化机制。换句话说，在你熟悉了Spring Framework，安

装好它以后，通过Spring进行持久化要比直接使用Hibernate容易很多。再看看例13-12。

例13-12：为了在Spring中使用而修改QueryTest类

```
package com.oreilly.hh;
import java.sql.Time;
import java.util.List;
import org.apache.log4j.Logger;
import com.oreilly.hh.dao.TrackDAO;
import com.oreilly.hh.data.Track;
/**
 *Retrieve data as objects
 */
public class QueryTest implements Test{
    private static Logger log=Logger.getLogger (QueryTest.class) ;
    private TrackDAO trackDAO;
    public void run () {
        //Print the tracks that will fit in five minutes
        List<Track> tracks=trackDAO.tracksNoLongerThan (
            Time.valueOf ("00: 05: 00") );
        for (Track track: tracks) {
            //For each track returned, print out the
            //title and the playTime
            log.info ("Track: \""+track.getTitle () +"\", "
                +track.getPlayTime () );
        }
    }
    public TrackDAO getTrackDAO () {return trackDAO; }
    public void setTrackDAO (TrackDAO trackDAO) {
        this.trackDAO=trackDAO;
    }
}
```

重新实现的QueryTest也定义了一个私有成员变量，用于引用TrackDAO对象。run () 方法调用trackDAO.tracksNoLongerThan () 方法，并为它传递了一个表示5分钟的Java.sql.Time类型的变量。这段

代码循环访问查询结果，用Log4J打印输出Track对象的title和playTime属性。最后看看例13-13。

例13-13：重新实现的AlbumTest

```
package com.oreilly.hh;
import java.sql.Time;
import java.util.*;
import org.apache.log4j.Logger;
import com.oreilly.hh.dao.*;
import com.oreilly.hh.data.*;
/**
 *Create sample album data, letting Hibernate persist it for us.
 */
public class AlbumTest implements Test{
    private static Logger log=Logger.getLogger (AlbumTest.class) ;
    private AlbumDAO albumDAO; ❶
    private ArtistDAO artistDAO;
    private TrackDAO trackDAO;
    public void run () {
        //Retrieve (or create) an Artist matching this name
        Artist artist=artistDAO.getArtist ("Martin L.Gore", true) ; ❷
        //Create an instance of album, add the artist and persist it
        //to the database.
        Album album=new Album ("Counterfeit e.p.", 1,
            new HashSet<Artist> () , new HashSet<String> () ,
            new ArrayList<AlbumTrack> (5) , new Date () ) ;
        album.getArtists () .add (artist) ;
        album=albumDAO.persist (album) ; ❸
        //Add two album tracks
        addAlbumTrack (album, "Compulsion", "vol1/album83/track01.mp3",
            Time.valueOf ("00: 05: 29") , artist, 1, 1) ;
        addAlbumTrack (album, "In a Manner of Speaking",
            "vol1/album83/track02.mp3", Time.valueOf ("00: 04: 21") ,
            artist, 1, 2) ;
        //persist the album
        album=albumDAO.persist (album) ; ❹
        log.info (album) ;
    }
    /**
     *Quick and dirty helper method to handle repetitive portion of
     creating
```

```

    *album tracks.A real implementation would have much more
flexibility.
    */
    private void addAlbumTrack (Album album, String title, String
file,
    Time length, Artist artist, int disc,
    int positionOnDisc) {
        //Create a new Track object and add the artist
        Track track=new Track (title, file, length, new HashSet<Artist>
() ,
        new Date () , new StereoVolume () , SourceMedia.CD,
        new HashSet<String> () ) ;
        track.getArtists () .add (artist) ;
        //Persist the track to the database
        track=trackDAO.persist (track) ;
        //Add a new instance of AlbumTrack with the persisted
        //album and track objects
        album.getTracks () .add (new AlbumTrack (track, disc,
positionOnDisc) ) ;
    }
    public AlbumDAO getAlbumDAO () {return albumDAO; }
    public void setAlbumDAO (AlbumDAO albumDAO) {
        this.albumDAO=albumDAO;
    }
    public ArtistDAO getArtistDAO () {return artistDAO; }
    public void setArtistDAO (ArtistDAO artistDAO) {
        this.artistDAO=artistDAO;
    }
    public TrackDAO getTrackDAO () {return trackDAO; }
    public void setTrackDAO (TrackDAO trackDAO) {
        this.trackDAO=trackDAO;
    }
}
}

```

AlbumTest比CreateTest和QueryTest都要更复杂，因为它要处理多个对象的创建和持久化，以及关联效果。可以逐步看看它的代码：

❶就像CreateTest和QueryTest一样，AlbumTest类也定义了一系列私有字段来引用它需要的所有DAO对象：trackDAO、artistDAO以及albumDAO。

❷ AlbumTest首先使用artistDAO.getArtist () 取回一个Artist对象, 如果这个方法没有找到请求的艺人对象, 就会创建一个新的Artist对象。

❸持久化Album实例。这一步会在数据库中创建一行数据, 并返回一个具有非null值id属性的Album对象。我们现在正在持久化Album记录, 这样就能够使用新的Album实例来创建多个Track对象, 再把它们关联到这个新的Album对象。为了让这一步能够正常运行, 需要确保我们的Album和Track对象均具有非null的id属性。

❹接着再增加一系列Track对象。要创建Track对象, 我们首先创建一个新的Track实例, 再用trackDAO.persist () 方法来持久化该Track对象。在addAlbumTrack () 方法中, 我们创建了几个Track对象, 再将它们与Album组合起来, 放到AlbumTrack关系对象中。Album上的tracks属性有一个一对多关系, 它的cascade属性设置为CacascadeType.ALL, 所以当我们再次持久化专辑对象时, 它会自动在ALBUM_TRACKS表中创建相应的数据行。

这就是我们对Test接口的实现。所有通用的处理已经转换到了所有DAO的持久化代码中, 接着再把我们单独的CreateTest、QueryTest以及AlbumTest类移植到其属性引用了这些DAO的bean中, 同时将实际的测试代码移植到Test接口要求的run () 方法中。这样, Spring就可

以将所有这些组件串接在一起。下一节我们将看看如何执行这些测试类。

TestRunner: 加载Spring ApplicationContext

如果我们没办法加载Spring的ApplicationContext，并执行我们的Test对象，前面的所有代码就无法使用。为此，我们要创建一个带有static main () 方法的TestRunner类，并从我们的Ant build.xml中调用该方法。例13-14完整地列出了TestRunner类的内容。这个类负责加载我们的Spring ApplicationContext，取回一个Test实现，并执行它。

例13-14: 加载Spring ApplicationContext

```
package com.oreilly.hh;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
/**
 *A simple harness to run our tests.Configures Log4J,
 *creates an ApplicationContext, retrieves a bean from Spring
 */
public class TestRunner{
private static Logger log;
public static void main (String[]args) throws Exception{
//Configure Log4J from a properties file
PropertyConfigurator.configure (
TestRunner.class.getResource ("/log4j.properties") ); ❶
log=Logger.getLogger (TestRunner.class) ;
//Load our Spring Application Context
log.info ("Initializing TestRunner.....") ;
log.info ("Loading Spring Configuration.....") ;
ApplicationContext context=❷
new ClassPathXmlApplicationContext ("applicationContext.xml") ;
```

```
//Retrieve the test name from the command line and
//run the test.
String testName=args[0];
log.info ("Running test: "+testName) ;
Test test= (Test) context.getBean (testName) ; ❸
test.run () ;
}
}
```

TestRunner负责为我们做三件事情，如JavaDoc中所示：

❶通过引用类路径的根目录下的log4j.properties来配置Log4J。

❷使用ClassPathXmlApplicationContext对象来创建一个Spring的ApplicationContext对象。ClassPathXmlApplicationContext的构造函数接受一个字符串参数，用这个参数来指明Spring XML配置文件在类路径中的位置。在这个例子中，我们的applicationContext.xml位于类路径的根目录（紧挨着log4j.properties文件）。

❸最后，我们从命令行参数中得到bean的名称，再从ApplicationContext中取回相应的Test对象。可以看到，从ApplicationContext中取回特定名称的bean是非常容易的，只需要调用context.getBean（名称），再将取回的结果转换为想要的类型。

运行CreateTest、QueryTest以及AlbumTest

为了运行TestRunner，并从我们的Spring ApplicationContext中取回正确的bean对象，还需要修改我们的Ant build.xml脚本。找到名为

ctest、qtest以及atest的构建目标，修改它们以包含以下XML，如例13-15所示。

例13-15：从Ant中执行TestRunner

```
<target name="atest"description="Creates and persists some
album data"
  depends="compile">
  <java classname="com.oreilly.hh.TestRunner"fork="true">
  <classpath refid="project.class.path"/>
  <arg value="albumTest"/>
  </java>
  </target>
  <target name="ctest"description="Creates and persists some
sample data"
    depends="compile">
    <java
classname="com.oreilly.hh.TestRunner"fork="true"failonerror="true">
    <classpath refid="project.class.path"/>
    <arg value="createTest"/>
    </java>
    </target>
    <target name="qtest"description="Runs a
query"depends="compile">
    <java classname="com.oreilly.hh.TestRunner"fork="true">
    <classpath refid="project.class.path"/>
    <arg value="queryTest"/>
    </java>
    </target>
```

TestRunner类使用它的第一个命令行参数作为要从Spring ApplicationContext获取的bean的名称。在build.xml中，我们在调用TestRunner时，就将bean（applicationContext.xml中的）的名称作为参数传递给它。

为了创建测试数据库，像平常那样运行`ant schema`；为了将数据插入到数据库中，则需要运行我们新版本的`ant ctest`：

```
%ant schema
%ant ctest
Buildfile: build.xml
prepare:
compile:
ctest:
[java]INFO TestRunner: 20-Initializing TestRunner.....
[java]INFO TestRunner: 21-Loading Spring Configuration.....
[java]INFO TestRunner: 25-Running test: createTest
BUILD SUCCESSFUL
Total time: 3 seconds
```

运行`ant qtest`，以调用新的`QueryTest`示例，并确认我们组装起来的所有东西是否可以正常工作：

```
%ant qtest
Buildfile: build.xml
prepare:
compile:
qtest:
[java]INFO TestRunner: 20-Initializing TestRunner.....
[java]INFO TestRunner: 21-Loading Spring Configuration.....
[java]INFO TestRunner: 25-Running test: queryTest
[java]INFO QueryTest: 25-Track: "Russian Trance", 00: 03: 30
[java]INFO QueryTest: 25-Track: "Video Killed the Radio Star",
00: 03: 49
[java]INFO QueryTest: 25-Track: "Test Tone 1", 00: 00: 10
BUILD SUCCESSFUL
Total time: 3 seconds
```

最后，我们可以运行新的`AlbumTest`示例。输入`ant atest`命令，你应该能够看到以下输出内容：

```
%ant atest
Buildfile: build.xml
prepare:
compile:
atest:
[java]INFO TestRunner: 16-Initializing TestRunner.....
[java]INFO TestRunner: 17-Loading Spring Configuration.....
[java]INFO TestRunner: 21-Running test: albumTest
[java]INFO AlbumTest: 40-Persisted Album: 1
[java]INFO AlbumTest: 59-Saved an album named Counterfeit e.p.
[java]INFO AlbumTest: 60-With 2 tracks.
BUILD SUCCESSFUL
Total time: 2 seconds
```

一切正常，现在还要做什么

Spring Framework和Hibernate彼此取长补短，配合得相当默契。如果你准备在一个大型应用程序中采用Hibernate，则应该考虑在Spring Framework的基础上来构建你的应用程序。在你投入时间学会了这种框架以后，我们相信你会发现为事务处理、连接管理以及Hibernate Session管理而编写的代码数量一定会有所减少。在这些常规任务上花费的时间越少，就可以投入更多的时间到你的应用程序特定的需求和业务逻辑处理上。可移植性（portability）是使用Spring（或任何类似的IoC容器）和DAO模式的另一个原因。虽然Hibernate是目前众多持久化库中的首选，但也说不定在未来的10年中又会冒出什么新技术。如果你将Hibernate特定的代码与应用程序的其他部分隔离开来，万一要试验下一种什么新技术时，就方便多了。

注意：Spring确实可以负责许多乏味的工作。但这不应该成为我们不去学习Hibernate细节的一个借口。

当和Spring配合使用时，小心不要被Hibernate的简单性所迷惑。本书的几位作者一致同意，虽然Hibernate是一件非常好的东西，但有时也会因为某些原因而很难调试和诊断Hibernate：输入错了的一个字符、没有正确映射的数据表、稍微不正确的flush（刷新）模式或者是一些神秘的JDBC驱动程序的不兼容问题。Spring之所以让Hibernate变得容易，是因为它提供了一些实用的抽象，让你的操作变得更简单。但是这样的简单性减少了需要直接在数据库中执行SQL语句的机会，让你更加脱离底层细节。虽然你可能不必自己编写事务处理代码，但在遇到问题时，这些抽象也让你更难诊断出产生错误的根本原因。不要误解，我是不会脱离Spring而单独使用Hibernate的，但是如果你牢固地掌握好了Hibernate的底层细节，那么就能更快地诊断出问题是出在了哪儿。

在下一章，我们将向你展示更高级的技术——如何将Hibernate集成到一个称为Stripes的Web应用程序框架中。在这个Web程序框架中，你将看到如何将Spring作为Stripes和Hibernate之间一个中立的代理。在你阅读学习下一章时，你应该牢记一个事实：目前使用的大多数流行的Web应用程序框架都提供了与Spring直接集成的某种功能。如果你使用的是Struts 2、Wicket或者Spring MVC，它们中的许多概念是相同的。

Spring是软件的Rosetta.Stone（译^[1]），在你接受它以后，就可以访问为与Spring集成而设计的所有开发库。以Spring作为基石，你就可以随着需求的改变而更容易地在不同技术之间进行转换。比如，不用Java，而是用JRuby或Groovy来编写DAO组件；使用Quartz来集成cron-like表达式；以及使用像Apache CXF之类的库将服务对象发布为SOAP服务端点等。

^[1] 美国Rosetta Stone（罗赛塔石碑语言学习软件）是风靡世界的多媒体英语教学软件。

第14章 画龙点睛：用Stripes集成Spring和Hibernate

在最近的几年中，各种Java Web框架如雨后春笋快速兴起。过去一段时间内，Struts被认为是事实上的Web应用程序的Java框架，但现在人们意识到还有各种选择可以使用。Java Server Faces（JSF）在企业空间中占有一定的份额，Spring MVC随Spring Framework也安装到许多应用中，不过，发现Stripes的开发人员也会经常选择这种框架。Stripes的知名度虽然没有Spring那么大，但是众所周知，市场成功并不总是直接由品质决定的。Stripes就是那种默默无闻，却又做出了很多非同寻常的成果的项目之一。

如果你对某种Web开发框架很有经验，可能会注意到有很多方法可以将Java代码和URL以及表单提交绑定起来。这些方法中的大多数都需要用复杂的XML和Java代码来做些非同寻常的处理，它们如此复杂和难以使用，以至于很多人放弃使用Java作为Web应用程序的开发工具，因为这将以牺牲实现速度作为代价。放弃Java框架，也就错过了已经用Java开发的众多优秀的开发框架，也与这种功能丰富的开发语言失之交臂。我们的感觉是Java提供的东西非常多，然而Stripes通过充分利用Java的功能和一致的体系结构，解除了以往Java Web开发中的诸多痛苦。虽然大多数开发人员会有更好的决断，但Struts在相当

长的一段时间内垄断了Java Web框架。Tim Fennell之所以要创建Stripes，就是为了取代Struts Web框架，因为他不喜欢将所有东西都放在struts-config.xml中，更不喜欢为了完成简单的任务还得管理很多配置文件（[\[1\]](#)）。他以Java 5和Servlet 2.4作为项目的起点，就能够对Java Web开发的现状进行一定的改进。对于原来Struts中大部分繁琐的任务，Stripes则通过合理的默认值、反射（reflection）、标注、基于泛型的类型推导（type inference）来加以简化。结果，Stripes就成为一种简洁、易于理解和扩展的开发框架，让Java Web开发变成了一件有趣的事。

安装Stripes

Stripes项目（[\[2\]](#)）的目标就是要简化开发人员的生活。为此，对配置提出一定的约定，当应用程序广泛使用默认设置时，也有办法可以修改这些默认设置。就像在前面介绍Spring的那章一样，我们并不打算完整地介绍Stripes为开发人员提供的所有友好的功能，只是希望你可以通过我们的介绍而明白Stripes是什么，以及如何让它同Spring和Hibernate协同工作。作为开始，最好是先了解几个与Stripes应用程序有关的概念。虽然你只需要负责与ActionBean和视图打交道，也应该明白Stripes实现的DispatcherServlet和StripesFilter的工作原理。

DispatcherServlet

在Stripes应用程序中，通常只有一个J2EE HttpServlet接口的实现，由Stripes的Dispatcher-Servlet为你提供这个实现。DispatcherServlet监听到来的URL请求，再决定应该实例化哪个ActionBean，以及应该调用那个ActionBean上的什么方法。可以将Dispatcher看做是应用程序的“管理器”。当请求到达应用程序时，Dispatcher会检查请求，再决定应该将这个请求委托给程序的哪部分负责处理。按照约定，一般将DispatcherServlet映射到*.action URL。

StripesFilter

StripesFilter封装了程序要处理的所有HTTP请求。当直接请求一个JSP时，StripesDispatcher就不会有机会运行，这时就得由StripesFilter负责为JSP和ActionBean提供一些Stripes的特定功能，二者的处理方式差不多是一样的。StripesFilter可以执行对multipart类型的表单的处理、本地化选择、flash scope管理（[\[3\]](#)）以及last stop异常处理。

ActionBean

当编写ActionBean时，就可以真正看到Stripes的可贵之处。这个接口只需要为一个名为context的属性配备一个getter和setter方法，context将是ActionBeanContext的一个实例。DispatcherServlet使用ActionBean上的反射，以及HTTP请求参数和ActionBean中的标注，就可以决定应该运行什么方法。

除了 `setContext ()` 和 `getContext ()` 方法，`ActionBean` 还可以包含几个返回 `Resolution` 的方法，以及用于同视图进行交互的属性存取器（`accessor`）。稍后我们开始构建示例时，你就会明白这里说的这些概念是什么意思了。你的 `ActionBean` 不必进行类似 `HttpRequest.getParameter ()` 之类的调用，因为 `Stripes` 可以自动负责将请求参数绑定到对象。

视图

`Stripes` 使用 JSP 作为它的视图技术。当 `ActionBean` 将请求转发到 JSP 时，`Stripes` 会为 JSP 提供一个对该 `ActionBean` 的引用，以便可以用 JSTL 表达式语言来取出 `ActionBean` 中的数据。反之，在 JSP 中可以用 `useActionBean` 标签来调用某个 `ActionBean` 的事件处理器，以便为显示准备请求（例如通过格式化属性）。`Stripes` 也提供了一个简单的 JSP 标签库，来帮助链接应用程序的各个部分和提供表单。

[1] 参见 <http://stripesframework.org/display/stripes/Stripes+vs.+Struts>.

[2] <http://stripesframework.org/>.

[3] `flash scope` 是一个概念，其本质是临时储存一些属性给（并且仅给）下一个请求使用，使用过后便被清除掉。

准备Tomcat

我们假设你已经有了一个可以跑起来的Apache Tomcat环境，还需要确保拥有Tomcat环境的manager（管理员）角色的用户身份。可以修改这些安全配置的地方是\$CATALINA_HOME/conf/tomcat-users.xml，请按例14-1所示的内容来调整你的tomcat-users.xml文件。

例14-1：在tomcat-users.xml中定义一个manager角色

```
<?xml version='1.0'encoding='utf-8'?>
<tomcat-users>
<role rolename="manager"/>
<role rolename="tomcat"/>
<role rolename="role1"/>
<user username="tomcat"password="tomcat"roles="tomcat,
manager"/>
<user username="both"password="tomcat"roles="tomcat, role1"/>
<user username="role1"password="tomcat"roles="role1"/>
</tomcat-users>
```

如果在tomcat-users.xml中新添加了一个manager角色，则需要保存该文件，并重新启动Tomcat。接下来就可以开始创建Stripes应用程序了。

创建Web应用程序

既然Tomcat实例已经可以跑起来了，接下来就开始创建一个Web应用程序。首先，要在你的项目目录中为我们的Web应用程序创建目录结构，如例14-2所示。你可以自己手工创建这个目录，也可以从本书的网站下载代码示例。

例14-2：创建Web应用程序目录结构的命令

```
$mkdir -p webapp/WEB-INF
```

作为开始，我们要在应用程序中加入一个web.xml和index.jsp文件，并部署它们。每个J2EE Web应用程序都需要一个web.xml文件，所以我们就先从这儿开始。稍后我们再用Filter和Servlet标签来配置这个文件，不过现在就把这个只有空架子的web.xml文件放在webapp/WEB-INF中了，如例14-3所示。

例14-3：最简单的webapp/WEB-INF/web.xml文件

```
<?xml version="1.0"encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"version="2.4">
</web-app>
```

一个Web应用程序至少要有有一个视图供用户查看，所以我们先在应用程序的根目录（webapp/）下添加一个非常简单的index.jsp文件。再一次，我们还不准备搞些花哨的风格样式；我们只需要页面上能有些内容，好让我们知道有东西在运行就可以了。代码如例14-4所示。

例14-4：一个简单的JSP文件webapp/index.jsp

```
<?xml version="1.0"?>
<%@page contentType="text/html; charset=UTF-8"language="java"%>
Hello World
```

目前将应用程序部署到Tomcat有许多种方法，你可以随意使用你喜欢的任何方法。我比较喜欢用Ant deploy（部署）构建任务将上下文信息发送到Tomcat。在应用程序上下文配置文件中可以指定docBase，告诉Tomcat到哪儿去找开发位置上的应用程序。采用这种部署技巧，你只需要部署一次应用程序。将程序部署到Tomcat可能会花不少时间，所以与每次都需要部署程序相比，采用这种办法以后，你的编译和测试周期将会变得更快。

为了运行应用程序，Tomcat还需要知道一些与之相关的信息。为Tomcat提供信息的一种方法就是将这些信息放在context文件中（如例14-5所示）。我们需要让Tomcat知道的主要事情就是应用程序的位置。

例14-5：一个tomcat-context.xml文件的例子

```
<?xml version="1.0"?>
<Context
docBase="/home/rfowler/current/examples/ch14/webapp"❶
debug="0"
reloadable="true"❷
>
</Context>
```

❶docBase属性用于指定应用程序将位于Tomcat服务器的文件系统的哪个位置。你需要将这个属性修改为你的计算机中应用程序的实际位置。

❷Context的reloadable属性用于指定Tomcat是否应该监视应用程序类文件的变化，并在有变化时重新加载应用程序的上下文。打开重新加载可以方便我们的开发周期，但是当应用程序发布为产品以后，则会浪费一定的CPU周期。

现在，我们已经编写好了一个context文件，接下来就更新build.xml文件，以便可以部署应用程序。在本章的构建文件中还需要添加几个依赖文件，我们打算将所有的修改内容都一次性复制到这里，如例14-6所示。

例14-6: build.xml中新添加的Tomcat依赖

```
.....
<artifact: dependencies pathId="dependency.class.path"
filesetId="dependency.fileset">
<dependency
groupId="hsqldb"artifactId="hsqldb"version="1.8.0.7"/>
<dependency groupId="mysql"artifactId="mysql-connector-java"
version="5.0.5"/>
```



```

    <dependency groupId="org.hibernate"artifactId="hibernate"
version="3.2.5.ga">
    <exclusion groupId="javax.transaction"artifactId="jta"/>
    </dependency>
    <dependency groupId="org.hibernate"artifactId="hibernate-tools"
version="3.2.0.beta9a"/>
    <dependency groupId="org.apache.geronimo.specs"
artifactId="geronimo-jta_1.1_spec"version="1.1"/>
    <dependency groupId="log4j"artifactId="log4j"version="1.2.14"/>
    <dependency
groupId="javax.servlet"artifactId="jstl"version="1.1.1"/>
    <dependency
groupId="taglibs"artifactId="standard"version="1.1.1"/>
    <dependency groupId="org.hibernate"artifactId="hibernate-
annotations"
version="3.3.0.ga"/>
    <dependency groupId="org.hibernate"
artifactId="hibernate-commons-annotations"
version="3.3.0.ga"/>
    <dependency groupId="org.springframework"artifactId="spring"
version="2.5"/>
    <dependency groupId="commons-dbcp"artifactId="commons-dbcp"
version="1.2.2"/>
    <dependency
groupId="net.sourceforge.stripes"artifactId="stripes"
version="1.4.3"/>❶
    <dependency groupId="tomcat"artifactId="servlet-
api"version="5.5.12"/>❷
    <dependency groupId="tomcat"artifactId="catalina-ant"
version="5.5.15"/>❸
    <dependency groupId="tomcat"artifactId="jasper-compiler"
version="5.5.15"/>
    <dependency groupId="tomcat"artifactId="jasper-runtime"
version="5.5.15"/>❹
    </artifact: dependencies>
.....

```

❶ Stripes这个artifact配置提供了使用Stripes框架所需要的jar库。

❷ servlet-api这个artifact配置提供了包含J2EE Servlet接口和支持类的库。HttpServletRequest和HttpServletResponse类都是由这个artifact提供的。

❸catalina-ant artifact Id提供了一些用于同运行的Apache Tomcat实例进行交互的Ant构建任务。简单来说，你将在构建文件中增加一个新的构建目标，它会用到来自这个artifact的Tomcat的deploy构建任务。

❹jasper-compiler和jasper-runtime这两个artifact，除了catalina-ant以外，Tomcat的deploy标签也需要它们。

到这以后，下一步就是用taskdef建立Catalina Ant构建任务，为部署应用程序定义一个新的target。Ant的deploy构建任务将为Tomcat发送一个tomcat-context.xml文件（例14-5），包括身份认证信息和context路径，如例14-7所示。

例14-7：用于部署应用程序的Ant构建目标

```
.....
<taskdef name="hibernatetool"
classname="org.hibernate.tool.ant.HibernateToolTask"
classpathref="project.class.path"/>
<!-- Teach Ant how to use Tomcat's deploy task -->
<taskdef name="deploy" classpathref="dependency.class.path"
classname="org.apache.catalina.ant.DeployTask"/>
<target name="db" description="Runs HSQLDB database management UI
against the database file--use when application is not running">
<java classname="org.hsqldb.util.DatabaseManager"
fork="yes">
<classpath refid="project.class.path"/>
<arg value="-driver"/>
<arg value="org.hsqldb.jdbcDriver"/>
<arg value="-url"/>
<arg value="jdbc: hsqldb: ${data.dir}/music"/>
<arg value="-user"/>
<arg value="sa"/>
</java>
</target>
.....
```

```
<target name="qtest3" description="Retrieve all mapped objects"
depends="compile">
  <java
classname="com.oreilly.hh.QueryTest3" fork="true" failonerror="true">
  <classpath refid="project.class.path"/>
  </java>
</target>
<target name="deploy">
  <deploy url="http://localhost: 8080/manager"❶
username="tomcat" password="tomcat"❷
path="/stripesapp"❸
config="${basedir}/tomcat-context.xml"/>❹
</target>
.....
```

❶deploy构建任务的url属性用于指定Apache Tomcat提供的manager servlet的URL。

❷username和password属性指定管理器角色的用户身份验证信息，这些信息是在例14-1中配置的。

❸path属性告诉Tomcat，应用程序应该部署到什么上下文路径。

❹config属性指定将要发送到Tomcat的上下文文件。这个文件如例14-5所示。

所有安排妥当以后，就应该可以运行ant deploy命令（如例14-8所示），再在Web浏览器中访问我们简单的Web应用程序了。

例14-8：部署应用程序

```
$ant deploy
Buildfile: build.xml
```

```
deploy:
[deploy]OK-Deployed application at context path/stripesapp
BUILD SUCCESSFUL
Total time: 3 seconds
```

上面示例输出中的"OK"表示Tomcat已经接受了新的应用程序。在浏览器中访问<http://localhost:8080/stripesapp>，看看这个程序是否存在，运行的对不对。在默认情况下，Tomcat将会查找和返回Web应用程序根目录下的index.jsp文件。因此，应用程序应该在浏览器中显示"Hello World"，如图14-1所示。

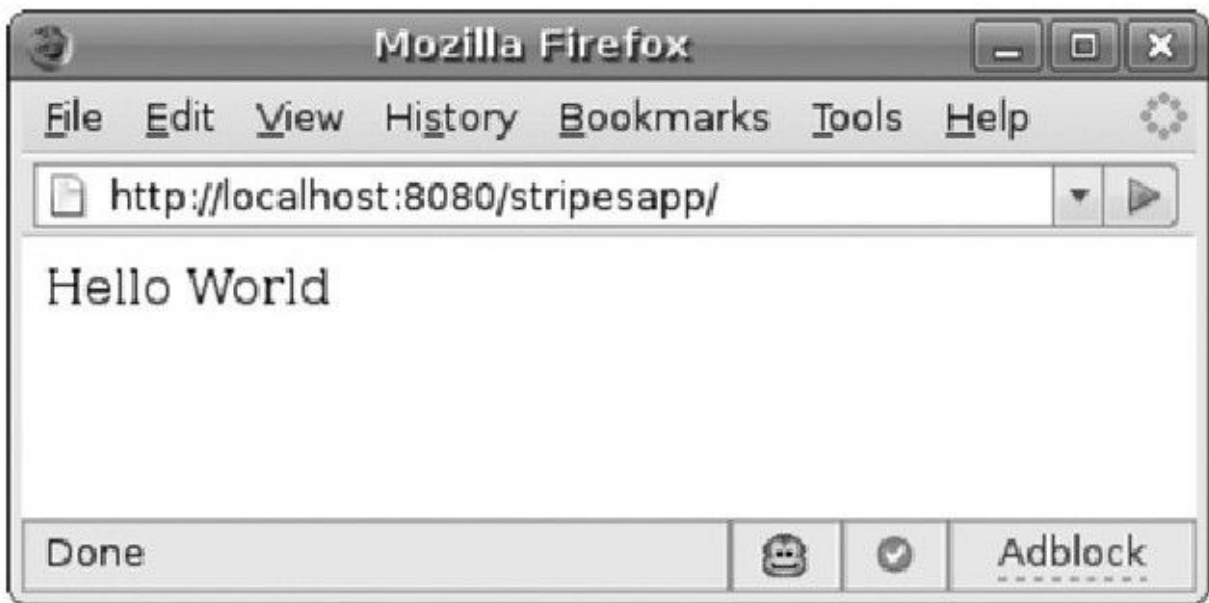


图 14-1 Tomcat显示"Hello World"

发生了什么事

虽然我们刚才进行的所有任务和**Stripes**或**Hibernate**没有任何关系，但是这些操作为使用**Stripes**铺平了道路。现在我们有了一个可以跑起来的**Apache Tomcat**安装实例，以及一个我们自己创建的运行良好的**Web**应用程序了。

增加Stripes

为了让我们的项目可以在Web环境中运行，需要对compile构建任务进行一些修改。到现在为止，我们一直在用Ant来启动应用程序，所以要为Ant提供正确的代码类路径信息。目前一切运行正常，但是Tomcat的类加载管理机制要复杂得多，因此，将所有需要的依赖文件直接复制到应用程序的WEB-INF/lib目录下是最简单的方法。我们也希望编译任务可以将文件放在WEB-INF中，这样Tomcat就可以直接找到它们。参见例14-9。

例14-9：针对Web应用程序而修改Compile构建任务

```
.....
<property name="source.root"value="src"/>
<property name="class.root"value="webapp/WEB-INF/classes"/> ❶
<property name="data.dir"value="webapp/WEB-INF/data"/> ❷
.....
<target name="compile"depends="prepare"
description="Compiles all Java classes">
  <javac srcdir="${source.root}"
  destdir="${class.root}"
  debug="on"
  optimize="off"
  deprecation="on">
    <classpath refid="project.class.path"/>
    </javac>
    <filter token="docroot"value="${basedir}/webapp"/> ❸
    <copy todir="webapp/WEB-INF"filtering="true"overwrite="true">
      <fileset dir="src"includes="applicationContext.xml"/>
    </copy> ❹
    <copy todir="webapp/WEB-INF/lib"flatten="true">
      <fileset refid="dependency.fileset"/>
    </copy> ❺
```

</target>

.....

❶ 因为我们正在将应用程序移植到J2EE Web应用程序，所以需要把Java类放到webapp/WEB-INF/classes目录下。最简单的实现方法就是修改class.root属性值。

❷ 由于当前的工作目录还不能确定，所以HSQLDB也不能够在数据的相对路径中找到数据库。因此，我们需要为Hibernate提供数据库的完整路径。为此，创建一个data.dir属性，当复制applicationContext.xml文件时会用到这个属性。

❸ ant的filter构建任务指定当复制applicationContext.xml时，需要用Web应用程序的实际路径来替换@docroot@标记。

❹ 需要将Spring的applicationContext.xml文件复制到webapp/WEB-INF目录下，以便Tomcat可以找到它。

❺ 既然应用程序不再由Ant来启动了，我们也就不能再依赖本地Maven仓库中找到的Java库。J2EE规范要求把Web应用程序的JAR文件放到WEB-INF/lib目录中，所以这一步就是将我们需要的文件从Maven本地仓库复制到WEB-INF/lib目录中。

为了让用docroot配置的过滤器（filter）对任何目录都有效，我们需要在src/application-Context.xml（第13章中创建的）中添加

@docroot@标记，例14-10演示了这一修改。

例14-10: applicationContext.xml中数据库配置文件位置的变化

```
.....
<property name="hibernateProperties">
  <props>
    <prop key="hibernate.show_sql">true</prop>
    <prop key="hibernate.format_sql">true</prop>
    <prop key="hibernate.transaction.factory_class">org.hibernate.
transaction.JDBCTransactionFactory</prop>
    <prop key="hibernate.dialect">org.hibernate.dialect.HSQLDialect
</prop>
    <prop key="hibernate.connection.autocommit">false</prop>
    <prop key="hibernate.connection.release_mode">after_transaction
</prop>
    <prop key="hibernate.connection.shutdown">true</prop>
    <prop key="hibernate.connection.driver_class">
org.hsqldb.jdbcDriver
    </prop>
    <prop key="hibernate.connection.url">jdbc: hsqldb:
@docroot@/WEB-INF
    /data/music</prop>
    <prop key="hibernate.connection.username">sa</prop>
    <prop key="hibernate.connection.password"></prop>
    <prop key="hibernate.current_session_context_class">thread
</prop>
    <prop key="hibernate.jdbc.batch_size">0</prop>
  </props>
</property>
.....
```

现在，我们的构建环境已经修改好，可以开始使用Stripes了。为了让Spring、Hibernate以及Stripes协同工作，还需要对web.xml进行多处修改。例14-11中演示了很多内容，但每个配置项都有一定的作用。稍后将介绍一下其中重要的配置。

例14-11：针对Stripes集成而修改后的web.xml

```
<?xml version="1.0"encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
version="2.4">
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class> ❶
  </listener>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
  </context-param>
  <!--Hibernate OpenSession Filter-->
  <filter>
    <filter-name>hibernateFilter</filter-name> ❷
    <filter-class>
      org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
r
    </filter-class>
    <init-param>
      <param-name>singleSession</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>sessionFactoryBeanName</param-name>
      <param-value>sessionFactory</param-value>
    </init-param>
    <init-param>
      <param-name>flushMode</param-name>
      <param-value>ALWAYS</param-value>
    </init-param>
  </filter>
  <filter>
    <display-name>Stripes Filter</display-name> ❸
    <filter-name>StripesFilter</filter-name>
    <filter-class>
      net.sourceforge.stripes.controller.StripesFilter
    </filter-class>
    <init-param>
      <param-name>ActionResolver.PackageFilters</param-name>
      <param-value>com.oreilly.*</param-value>
```

```

</init-param>
<init-param>
<param-name>ActionResolver.UrlFilters</param-name>
<param-value>WEB-INF/classes</param-value>
</init-param>
<init-param>
<param-name>Interceptor.Classes</param-name>
<param-value>
net.sourceforge.stripes.integration.spring.SpringInterceptor,
net.sourceforge.stripes.controller.BeforeAfterMethodInterceptor
</param-value>
</init-param>
</filter>
<filter-mapping>
<filter-name>hibernateFilter</filter-name>
<url-pattern>*.jsp</url-pattern>
<dispatcher>REQUEST</dispatcher>
</filter-mapping>
<filter-mapping>
<filter-name>hibernateFilter</filter-name>
<url-pattern>*.action</url-pattern>
<dispatcher>REQUEST</dispatcher>
</filter-mapping>
<filter-mapping>
<filter-name>StripesFilter</filter-name>
<url-pattern>*.jsp</url-pattern>
<dispatcher>REQUEST</dispatcher>
</filter-mapping>
<filter-mapping>
<filter-name>StripesFilter</filter-name>
<url-pattern>*.action</url-pattern>
<dispatcher>REQUEST</dispatcher>
</filter-mapping>
<servlet>
<servlet-name>StripesDispatcher</servlet-name> ❹
<servlet-class>
net.sourceforge.stripes.controller.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>StripesDispatcher</servlet-name>
<url-pattern>*.action</url-pattern>
</servlet-mapping>
</web-app>

```

❶ContextLoaderListener过滤器用于初始化Web应用程序的Spring Framework。

❷hibernateFilter是由Spring提供的，用于获取一个封装了所有请求处理的Hibernate会话。使用这个功能，我们就可以不必再自己管理Hibernate会话。这一功能之所以精彩，是因为正确的会话管理可能是编写支持Hibernate的Web应用程序时最具技巧性的处理之一。

❸将Stripes Filter映射到*.action和*.jsp HTTP请求。它提供了调用JSP或ActionBeans时所需的一些基本表单处理和配置服务。

❹StripesDispatcher servlet映射到了*.action，负责决定应该调用哪个ActionBean中的哪个方法，以及处理由这些事件方法返回的Resolution。

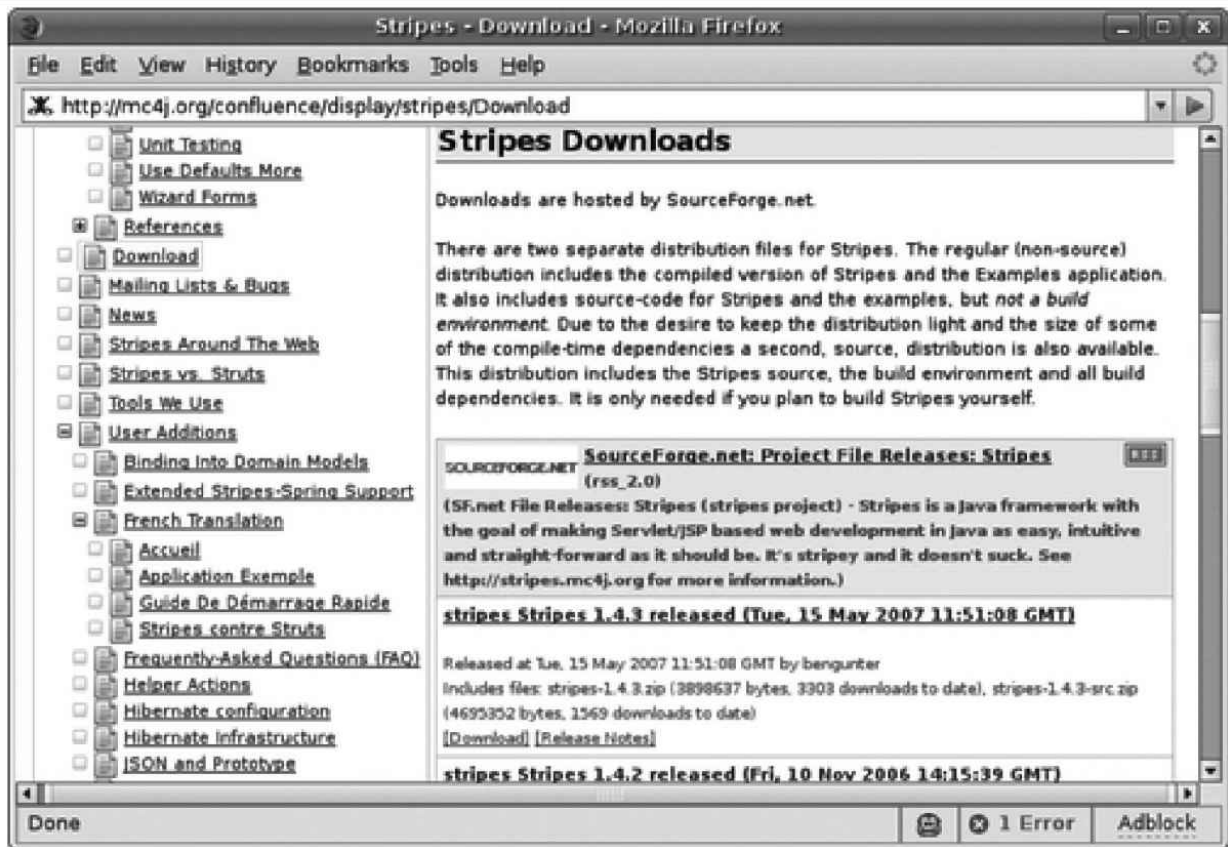


图 14-2 Stripes下载页面

在开始使用Stripes以前，为了让它正常运行，还有最后一个需要安装的配置。Stripes-Resources.properties文件位于classes类目录中，它为Stripes提供一些必需的格式化字符串。对于这个例子来说，最简单的办法就是用下载的随书示例代码，直接从examples/ch14/src目录中复制StripesResources.properties文件，不过也可以从Stripes下载包（[\[1\]](#)）中得到这个文件。在Stripes 1.4.3下载页面的中间，点击"Download"按钮，如图14-2所示，继续链接到SourceForge下载页面，对SourceForge我们都应该很熟悉了。在下载好stripes-1.4.3.zip以后，将它进行解压，再把stripes-1.4.3/lib/StripesResources.properties复制到你的src目录下。如

果愿意，你可以看看这个文件的内容，不过它只是本章示例需要的一个依赖文件而已。

最后就应该编写一点代码了。首先，我们来编写两个JSP文件，再接着开发一个ActionBean。如果你最近几年写过些JSP代码，那么我们在这儿编写的JSP文件也不会令你感到陌生。不过，你可能不认得的是前缀（prefix）为"stripes"的几个标签。Stripes标签库可以作为JSTL的补充，辅助你的应用程序类和视图协同工作。例14-12演示了一个用于编辑曲目专辑的页面源代码。

例14-12：曲目编辑视图：webapp/albums/edit.jsp

```
<%@page contentType="text/html; charset=UTF-8" language="java"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib
prefix="stripes" uri="http://stripes.sourceforge.net/stripes.tld"%>❶
<stripes: useActionBean
beanclass="com.oreilly.hh.web.AlbumActionBean"
var="actionBean" event="edit"/>❷
<h1>Album Edit Page</h1>
<stripes: form action="/Album.action">❸
<stripes: errors/>
<stripes: hidden name="album.id"></stripes: hidden>❹
<table>
<tr>
<td>Title: </td>
<td><stripes: text name="album.title"/></td>❺
</tr>
<tr>
<td>Discs: </td>
<td><stripes: text name="album.numDiscs"/></td>
</tr>
</table>
<h2>Album Comments</h2>
<c: choose>
<c: when test="${actionBean.album.id! =null}">
```

```
<stripes: link href="/albums/edit_comment.jsp">
<stripes: param name="album.id"value="${actionBean.album.id}"/>
Add A Comment
</stripes: link>
<c: if test="${empty actionBean.album.comments}">
There are no album comments yet.
</c: if>
</c: when>
<c: otherwise>
Please add the album before entering comments.
</c: otherwise>
</c: choose>
<ul>
<c: forEach items="${actionBean.album.comments}"var="comment">
<li>${comment}</li>
</c: forEach>
</ul>
<br/>
<stripes: submit name="save"value="Save"></stripes: submit>❹
</stripes: form>
```

可以看到，这是一个外观界面相当普通的JSP页面，不过，也有一些东西值得更仔细地研究一下：

❶taglib声明用于引入Stripes标签库，以便页面上的代码可以通过"stripes: "前缀来使用它们。

❷这里使用useActionBean标签来告诉Stripes初始化AlbumActionBean，如果它的edit事件没有发生的话（例如，浏览器直接请求JSP页面，而不是通过action URL），就运行这个事件。edit事件将会从数据库中加载Album对象，为生成表单做好准备。

❸Stripes的form标签会输出一个普通的HTML表单，以及很多隐藏在幕后的东西。它的action属性指定表单将要提交到的ActionBean。

④Stripes的hidden标签的作用类似于普通HTML的input标签。使用Stripes版本的标签的好处是：它的值是自动生成的。

⑤Stripes的text标签可以创建一个文本输入字段（这和你想到的应该一致）。hidden标签，它可以自动生成其value属性的值。

⑥Stripes的submit标签会生成一个典型的提交按钮。这里要注意的是：submit标签的name属性值是当表单提交时要调用的ActionBean事件处理方法的名称。在这个例子中，调用的是AlbumActionBean.save（）（本章后面在讨论ActionBean时将进一步解释事件和它们的处理器）。

Stripes也有一个label标签，可以帮助设置本地化（localization）配置，让代码保持简洁。我们在这没有使用它，是因为想只关注Hibernate的东西。幸好Stripes也为这些标签提供了很详细的文档（[\[2\]](#)）。

这个页面写好以后，接下来还要编写一个用于列出数据库中的专辑的页面，可以将这个页面作为一个加载页面，通过它能够看到数据库中有什么信息。例14-13中的页面代码看起来比edit.jsp页面要少一点，不过，要注意一下结尾处的Stripes link标签。

例14-13：专辑列表视图：webapp/albums/list.jsp

```
<%@page contentType="text/html; charset=UTF-8" language="java"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="stripes"
```

```
uri="http://stripes.sourceforge.net/stripes.tld"%>
<stripes: useActionBean
beanclass="com.oreilly.hh.web.AlbumActionBean"
var="actionBean"event="list"/>
<table>
<tr>
<th>title</th>
<th>discs</th>
<th>action</th>
</tr>
<c: forEach items="${actionBean.albums}"var="album">
<tr>
<td>${album.title}</td>
<td>${album.numDiscs}</td>
<td><stripes: link href="/albums/edit.jsp">
<stripes: param name="album.id"value="${album.id}"/>
edit
</stripes: link></td>
</tr>
</c: forEach>
</table>
<stripes: link href="/albums/edit.jsp">new</stripes: link>
```

`stripes: link`标签用于将程序和HTML锚点（anchor）链接起来，它提供了几个属性，可以构建指向ActionBean事件处理器以及JSP的URL。

编写ActionBean

接下来应该编写一个ActionBean。可以将ActionBean看做是MVC（Model、View、Controller）模式中的控制器组件。例14-14演示了我们的第一个AlbumActionBean，它可以让你很好地了解编写ActionBean需要涉及哪些内容。

这个类中的方法可以分为两大类：属性存取器和事件处理器。属性存取器就像其他Java Bean的setter和getter方法一样，所以它们看起来也差不多。另一方面，那些返回Resolution的方法就显得有些陌生了，不过其概念也相当简单。当一个请求到达StripesDispatcher时，HTTP请求中的某个部分就可以指示出一个事件的名称，由该事件的处理器负责处理这个HTTP请求。当Stripes请求的生命周期经过它的BindingAndValidation阶段后，就会调用由请求确定的事件处理器方法（StripesDispatcher使用反射来查找其名称和事件匹配的方法，该方法返回的就是一个Resolution）。由事件处理器返回的Resolution对象接着再由StripesDispatcher进行处理（通常进行转发或重定向）。例14-14演示了我们的AlbumActionBean.java。

例14-14：专辑控制器：AlbumActionBean.java

```
package com.oreilly.hh.web;
import java.util.List;
import org.apache.log4j.Logger;
import net.sourceforge.stripes.action.*;
import net.sourceforge.stripes.integration.spring.SpringBean;
import net.sourceforge.stripes.validation.*;
import com.oreilly.hh.dao.AlbumDAO;
import com.oreilly.hh.data.Album;
/**
 *Class that implements the web based front end of our Jukebox.
 *
 */
public class AlbumActionBean implements ActionBean{
/**
 *Logger
 */
private static Logger log=Logger.getLogger
(AlbumActionBean.class) ;
```

```

/**
 *The ActionBeanContext provided to this class by Stripes
DispatcherServlet.
 */
private ActionBeanContext context;
/**
 *The list of Album objects we will display on the Album list
page.
 */
private List<Album> albums;
/**
 *The Album we are providing a form for on the edit page.
 */
private Album album;
/**
 *The Data Access Object for our Albums.
 */
private AlbumDAO albumDAO;
public ActionBeanContext getContext () {❶
return context;
}
public void setContext (ActionBeanContext aContext) {
context=aContext;
}
/**
 *The default event handler that displays a list of Albums.
 *@return a forward to the Album list jsp.
 */
@DefaultHandler
public Resolution list () {❷
albums=albumDAO.list () ; ❸
return new ForwardResolution ("/albums/list.jsp") ;
}
/**
 *The event handler for handling edits to an Album
 *@return a forward to the Album edit jsp.
 */
public Resolution edit () {
if (album!=null) {❹
album=albumDAO.get (album.getId () ) ;
}
return new ForwardResolution ("/albums/edit.jsp") ;
}
/**
 *The event handler for saving an Album.
 *@return a redirect to the Album list jsp.
 */
public Resolution save () {

```

```

albumDAO.persist (album) ;
log.debug ("Redirecting to list! ") ;
return new RedirectResolution ("/albums/list.jsp") ;
}
/**
 *A getter for the view to retrieve the list of Albums.
 * @return a list of Albums
 */
public List<Album>getAlbums () {❶
return albums;
}
/**
 *A setter for the DispatcherServlet to call that provides the
album to
 *save.
 */
@param anAlbum
@ValidateNestedProperties ({❷
@Validate (field="title", required=true, on={"save"}) ,
@Validate (field="numDiscs", required=true, on={"save"})
})
public void setAlbum (Album anAlbum) {
log.debug ("setAlbum") ;
album=anAlbum;
}
/**
 *A getter for the edit view to call.
 * @return an Album
 */
public Album getAlbum () {
return album;
}
/**
 *A method Spring will call that provides this class with an
AlbumDAO
 *instance. @param anAlbumDAO The AlbumDAO object
 */
@SpringBean ("albumDAO") ❸
public void injectAlbumDAO (AlbumDAO albumDAO) {
this.albumDAO=albumDAO;
}
}

```

❶ ActionBean接口惟一要求必须实现的是setContext () 和
getContext () 方法， ActionBean可以通过这两个方法来获取有关它的

操作所在的Stripes环境的信息。

❷这个返回Resolution的public的方法在Stripes中称为事件处理器。它们被自动绑定到浏览器能够访问的URL上。例如，对于访问路径/stripesapp/Album.action?save=，Dispatcher就会选中这个方法。

❸现在还没有AlbumDao.list ()方法，所以我们需要将它增加到AlbumDAO接口（其定义位于第13章）和AlbumHibernateDAO实现类中。

❹“如果专辑不为null，那么就加载专辑”似乎有点逆向逻辑的意味，但是真正发生的情况是：当Stripes将请求绑定到ActionBean中的对象时，它看到的只是一个album.id参数，用这个id值来创建一个新的Album对象，接着再调用这个方法。但是我们真正想知道的是如何从数据库中加载一个Album对象，再编辑它，所以这就是我们在这里所做的处理。

❺当在请求中发现表单数据与bean属性的命名模型匹配时，Stripes就会调用ActionBean中相应的public类型的getter和setter方法。例如，当请求参数中有album.id、album.title以及album.numDiscs时，Stripes就会调用setAlbum ()方法。另一方面，当向浏览器生成表单时，就会使用getter方法预生成各个值（这也就是例14-12中的那些stripes: text之类的标签所完成的作用）。

⑥Stripes提供了验证标注，用于标明当调用事件处理器时应该看到哪些字段。我们在此处不准备深入介绍更详细的内容，你可以在Stripes的在线文档（[\[3\]](#)）中找到更多的信息。

⑦SpringBean标注是告诉Stripes：在Spring上下文中查找这个对象值，并插入到这里。我们在这里没有使用Spring应用程序中典型的public类型的setter方法，因为黑客（hacker）可能会调用这样的方法来正确地格式化Web请求，出于安全原因，我们应该防止类似的访问。为Spring要调用（[\[4\]](#)）的方法采用其他命名规范，通常是个好主意。

如前所述，AlbumDAO需要有一个list（）方法返回所有的Album对象，以及一个get（）方法根据专辑的id来取回对应的Album对象。为此，我们需要调整一下AlbumDAO接口和AlbumHibernateDAO实现。例14-15演示了更新后的AlbumDAO.java，修改过的部分以粗体突出显示。

例14-15：在AlbumDAO中增加list（）和get（）方法的定义

```
package com.oreilly.hh.dao;
import java.util.List;
import com.oreilly.hh.data.Album;
public interface AlbumDAO{
    public Album persist (Album album) ;
    public void delete (Album album) ;
    public List<Album>list () ;
    public Album get (Integer id) ;
}
```

例14-16以粗体突出显示了需要对AlbumHibernateDAO.java进行的修改。

例14-16: 在AlbumHibernateDAO中增加list () 和get () 方法的实现

```
package com.oreilly.hh.dao.hibernate;
import java.util.List;
import
org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import com.oreilly.hh.dao.AlbumDAO;
import com.oreilly.hh.data.Album;
public class AlbumHibernateDAO extends HibernateDaoSupport
implements AlbumDAO{
    public Album persist (Album album) {
        album= (Album) getHibernateTemplate () .merge (album) ;
        getSession () .flush () ;
        return album;
    }
    public void delete (Album album) {
        getHibernateTemplate () .delete (album) ;
    }
    @SuppressWarnings ("unchecked")
    public List<Album>list () {
        return getHibernateTemplate () .loadAll (Album.class) ;
    }
    public Album get (Integer id) {
        return (Album) getHibernateTemplate () .load (Album.class, id) ;
    }
}
```

例14-16中使用的是HibernateTemplate.loadAll () 方法，它与Session.loadAll () 的功能一样，将返回作为参数提供的类的所有持久化对象的列表。第13章已经讨论过使用HibernateTemplate类的优点。

现在我们已经有了两个视图，一个ActionBean，也对DAO进行了相应的修改。接下来就可以编译和体验这个Web应用程序了。相关命令如例14-17所示。

例14-17：编译我们的Stripes应用程序

```
$ant compile
Buildfile: build.xml
Overriding previous definition of reference to project.class.path
prepare:
compile:
[javac]Compiling 20 source files to
/home/rfowler/Hibernate Book/examples/ch12/webapp/WEB-INF/classes
[copy]Copying 1 file to/home/rfowler/Hibernate
Book/examples/ch12/webapp/
WEB-INF
BUILD SUCCESSFUL
Total time: 2 seconds
```

当部署例14-5所示的应用程序时，我们将Context元素的一个属性设置为：reloadable=true。这样设置以后，当为Web应用程序的WEB-INF/classes目录提供了新版本的类时，Tomcat将会自动重新加载程序的context。假设你让Tomcat仍然保持运行，如果稍等片刻，context就会自动加载完成。在这一处理完成以后，就可以用浏览器来访问 <http://localhost:8080/stripesapp/albums/list.jsp>，看到的页面应该如图14-3所示。

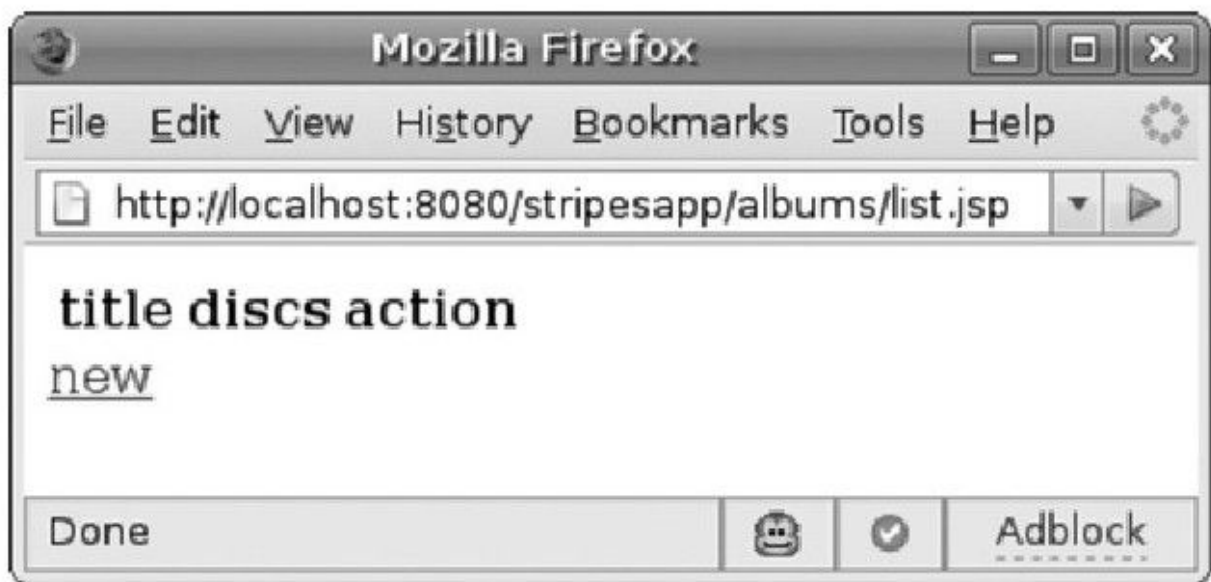


图 14-3 我们的ActionBean跑起来了

现在我们的数据库中还没有数据，所以这个页面也不会显示任何专辑信息。但是点击"New"链接，将会打开例14-12编写的编辑页面，如图14-4所示。



图 14-4 加载Edit（编辑）页面

可以想象到这个页面的功能，在表单中输入有效的数据，点击"Save"提交表单后，将会把数据保存到数据库中，再返回到列表视图，如图14-5所示。如果能够按这个流程走下来的话，就表明你已经成

功地集成了Hibernate、Stripes以及Spring！花些时间来思考一下本章进行的数据库处理。令人激动的应该还是没有花费多少代码就完成了这么多处理。你在AlbumDAO和AlbumHibernateDAO中添加了list（）和edit（）两个方法，使用AlbumDAO对象来加载和持久化Album对象。你也会注意到，整个过程中没有用任何代码来直接处理HTTPServletResponse或HttpServletRequest对象—Stripes已经为你处理好了这些繁琐的工作。

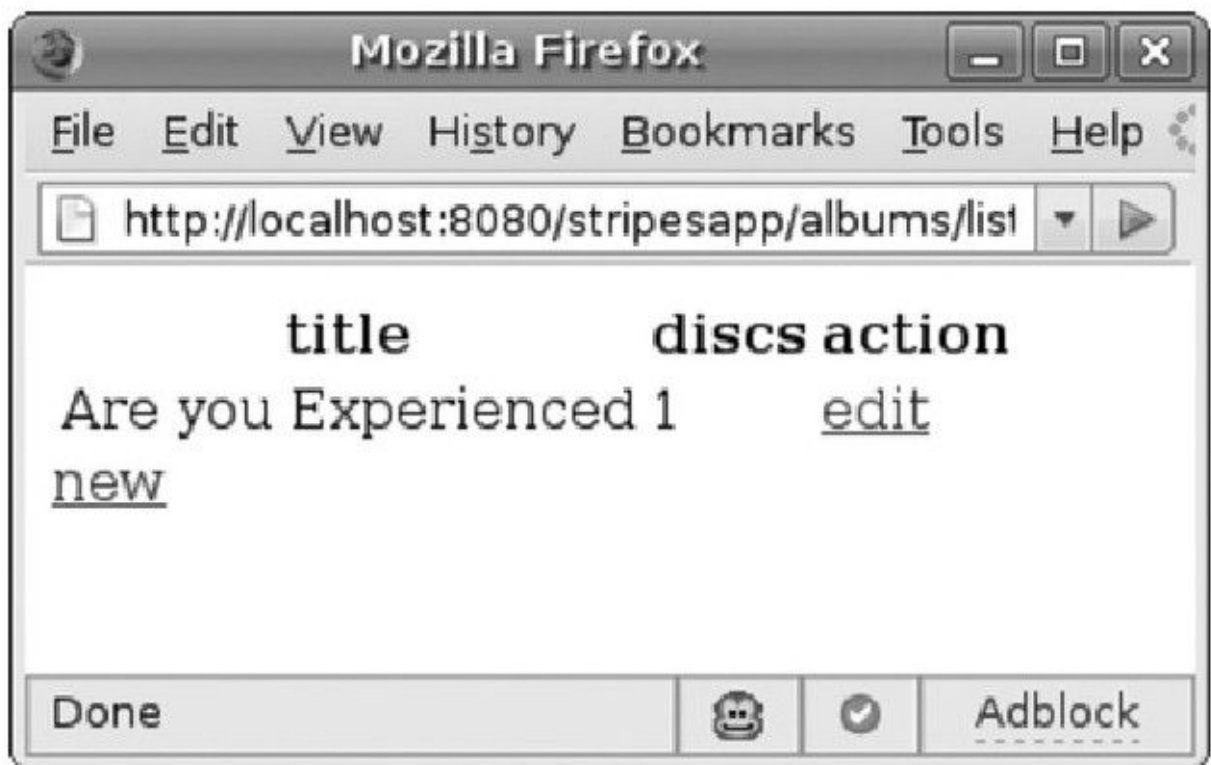


图 14-5 显示了一些内容的列表视图

刚才发生了什么

到目前为止，我们已经用Stripes、Spring以及Hibernate构建好了一个简单的Web应用程序。现在我们可以列出所有的专辑、创建新的专辑、对专辑进行编辑。AlbumActionBean使用Stripes和第13章写的Hibernate DAO来保存对象，为视图提供对象。

下一步要做什么

既然我们已经讨论了如何实现基本的插入、更新、显示数据的处理，我们就回过头来，看看怎么在程序中处理关联。现在我们还根本没有认真思考过它。

[1] <http://www.stripesframework.org/display/stripes/Download>.

[2] <http://www.stripesframework.org/display/stripes/Documentation>.

[3] <http://www.stripesframework.org/display/stripes/Validation+Reference>.

[4] <http://www.stripesframework.org/display/stripes/Spring+with+Stripes>.

处理关联

因为我们的例子还没有处理任何关联，所以就避开了典型应用程序中会遇到的一个复杂问题。其实，当DAO对象在它们的`persist ()`方法中调用`merge ()`方法时，就会持久化你发送给它的任何东西。如果正在持久化的对象没有包含所有与它关联的对象，那么就会改写原来持久化的数据，而那些没有包含的关联也会随之丢失。例如，如果一个专辑`Album`原本有两个评论，但是保存时使用的是一个空的评论集合，那么以前数据库中原有的评论就会丢失。我们在这一节就为添加和编辑专辑的评论而实现一个事件处理器和视图。只有这样，你才能明白我们正在解决的问题到底是怎么回事儿，我们再去讨论如何解决这个问题。首先，增加一个`editComments.jsp`文件，其内容如例14-18所示，它为添加专辑评论提供相应的表单。

例14-18：评论编辑器： `webapp/albums/edit_comment.jsp`

```
<%@page contentType="text/html; charset=UTF-8" language="java"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib
prefix="stripes" uri="http://stripes.sourceforge.net/stripes.tld"%>
<stripes: useActionBean
beanclass="com.oreilly.hh.web.AlbumActionBean" var=
"actionBean" event="edit"/>
<h1>Add a comment for the album<span style="font-style:
italic">${action
Bean.album.title}</span></h1>
<stripes: form action="/Album.action">
<stripes: hidden name="album.id"/>
```

```
Comment: <stripes: text name="comment"/>
<br/>
<stripes: submit name="saveComment" value="Save"/>
</stripes: form>
```

这个JSP文件看起来有点熟悉，因为它沿用了与例14-12同样的模式。你可以告诉stripes: form标签，这个页面要提交到我们已经创建的AlbumActionBean。由于stripes: submit标签的name属性为saveComment，所以我们需要在AlbumActionBean中增加一个public类型的saveComment () 方法，它同样也返回一个Resolution。因为stripes: text标签的name属性为comment，我们也需要在bean中创建setComment () 和getComment () 属性存取器。例14-19演示了AlbumActionBean.java中新增加的这些内容。

例14-19: AlbumActionBean.java中为了支持评论而新增加的内容

```
.....
/**
 *Event handler to save a comment to the Album
 *@return redirect to the edit Album page.
 */
public Resolution saveComment () {
    Album a=albumDAO.get (album.getId () );
    a.getComments () .add (comment);
    albumDAO.persist (a);
    RedirectResolution r=new RedirectResolution
("/albums/edit.jsp");
    r.addParameter ("album.id", album.getId () );
    return r;
}
/**
 *@return the comment
 */
public String getComment () {
    return comment;
}
```

```
}  
/**  
 * @param aComment the comment to set  
 */  
public void setComment (String aComment) {  
    comment=aComment;  
}  
.....
```

当重新编译，并使用几下新功能以后，你可能会发现程序有个非常严重的bug。如果先为某个Album增加些评论，接着再更新同一Album的标题或唱片数量时，你会发现评论不见了。产生这个bug的原因是每次调用saveAlbum () 方法时，Stripes就会创建一个新的Album对象，我们接着再告诉Hibernate保存这个新对象。这个新对象还没有评论，但它确实有个ID，所以Hibernate会更新持久化的Album，让它没有评论。

保存具有关联的对象，可以用两种方法。一种是将inverse标志设置为true，这样当更新对象时，对关联所做的任何修改都将被忽略。不过，这种方法对专辑评论没有效果，因为评论只是我们的模型中的字符串，并不是完整的实体，所以在评论对象上没有setAlbum () 之类的方法可以调用。另一种解决这个问题的方法是在Stripes调用bean上的任何setter存取器方法之前，就真正从数据库中加载这个对象以及它所关联的所有对象。这样，在Stripes开始修改这个对象以前，关联的对象已经生成好了。这种方法听起来很不错，我们试一试。

也应该思考一下Stripes处理请求的生命周期。Stripes Lifecycle Documentation ([\[1\]](#)) 详细解释了各个阶段发生的事情，如果你渴望研究额外的信息，参考这些文档就可以。简单来说，在解析好事件处理器以后，有一个称为BindingAndValidation的阶段，用于调用ActionBean上的各setter方法。Stripes提供了一种“拦截器”（interceptor）机制，通过这种机制，我们可以在BindingAndValidation运行以前插入我们自己的处理代码。

拦截器：扩展Stripes的强大方法

为了告诉我们的拦截器在什么时候运行，可以在我们的ActionBean中创建自己的标注。这样，我们的扩展就很自然地集成到Stripes的其他工作方式中。

一些人不赞成使用拦截器和面向切面的编程（Aspect Oriented Programming, AOP），因为调试AOP软件一般都很困难。在一些AOP应用程序中，很难确定正在运行什么代码和代码运行的时间。不过，随着一些新功能的出现，如Java 5标注，用自文档化（self-documenting）的方法就可能做更多的事情（而不用依赖非语言的窍门），所以我鼓励你对Stripes的拦截器实现保持开放的思想。事实上，Stripes和Spring都非常有效地利用了这种方法。Stripes使用的拦截器模式会检查ActionBean中的某些标注，并基于那些标注而采取一定

的行动。采用这种方法，**Stripes**也可以支持基于**Spring**的依赖注入，还可以在适当的生命周期阶段调用特定的**ActionBean**方法。按照这种模式来增加我们自己的功能也相当简单，所以我们打算在绑定和验证阶段之前增加一种用于加载数据bean的机制。

如果你以前还没有写过标注，对它的语法可能会觉得有些陌生，但思想是相当直观的。例14-20中的代码是说，开发人员将能够用**@LoadBean**标注来标记方法，提供在**Binding-AndValidation**发生之前需要加载的成员的名称（标注本身并不提供这些语义，它只是支持这种语法。我们在例14-22中演示的**Interceptor**代码就使用该标注来实现想要的功能）。例14-20演示了我们的新标注，应该将它保存为**LoadBean.java**。

例14-20：创建LoadBean标注

```
package com.oreilly.hh.web;
import java.lang.annotation.*;
/**
 *An annotation used to mark methods with a bean to load.
 *
 *@author Ryan Fowler
 */
@Retention (RetentionPolicy.RUNTIME) ❶
@Target ({ElementType.METHOD}) ❷
@Documented❸
public@interface LoadBean{❹
/**
 *The name of the bean to load.
 */
String value () ; ❺
}
```

❶这个标注中的@RetentionPolicy取值为RunTime，这是在告诉Java，在编译生成的类中保存标注信息，以便我们的拦截器（Interceptor）在运行时能够看到它们。对，在编写标注类时也可以使用很多其他标注。

❷@Target标注用于指定LoadBean标注应该应用到什么类型的元素。当运行特定的ActionBean事件处理器时，我们的拦截器就会检查标注。由于事件处理器是ActionBean中的方法，所以这里的目标类型设置为ElementType.METHOD。

❸@Documented标注指定应该用JavaDoc来文档化LoadBean标注。

❹面对@interface这样的说明符，标注定义语法让人乍看起来感觉很不安。在Java中增加新的功能的愿望与日俱增，但是却没有增加新的关键字，这样就导致了在定义标注时还得重用interface关键字。要是这两种需求能够从技术上得以很好地解决，标注定义的可读性也就不会受到什么影响了。

❺value（）方法的定义只是规定这个标注只有一个参数，该参数的名称是value。之所以要使用value作为参数名称，原因是这样可以使用简写形式的@LoadBean（"memberName"），而不必使用稍微长些的@LoadBean（value="memberName"）。这是标注机制本身支持的一种规范。

现在，我们可以在AlbumActionBean中相关的事件处理器上应用这个标注，以告诉拦截器它需要做什么。@LoadBean标注稍后会告诉我们的拦截器在Stripes调用Album上的setter方法之前加载Album bean。

例14-21：新的AlbumActionBean.save () 事件处理器

```
.....
/**
 *The event handler for saving an Album.
 *@return a redirect to the Album list jsp.
 */
@LoadBean ("album")
public Resolution save () {
    albumDAO.persist (album) ;
    log.debug ("Redirecting to list! ") ;
    return new RedirectResolution ("/albums/list.jsp") ;
}
.....
```

OK，这一切听起来都相当不错，但是神奇的拦截器的工作原理是怎么样？为了加载那个Album bean，我们需要某个东西，它应该在BindingAndValidation阶段之前运行，这样才能加载数据。前面我们提到，Stripes提供了一种拦截器执行功能，这正是我们需要的钩子

（hook）。为了使用这个功能，我们得编写一个实现Stripes Interceptor接口的类，再用StripesFilter的init-param告诉Stripes要使用这个拦截器。在这个拦截器类中用@Intercepts标注可以告诉Stripes它对什么生命周期阶段感兴趣，在那些执行点上就会调用实现的intercept () 方法。

在我们的`intercept ()`方法中，会在事件处理器中检查`@LoadBean`标注。如果那个标注存在，就用Spring和Hibernate尝试加载那个标注的`value`属性指定的名称所对应的对象。这一步完成以后，通过调用`ExecutionContext.proceed ()`和它的返回值，拦截器就会指示Spring接下来继续要做什么事情。例14-22演示了`LoadObjectInterceptor.java`的源代码。

例14-22：对象加载拦截器

```
package com.oreilly.hh.web;
import java.lang.reflect.Method;
import javax.servlet.http.HttpServletRequest;
import org.apache.log4j.Logger;
import org.springframework.orm.hibernate3.HibernateTemplate;
import net.sourceforge.stripes.action.Resolution;
import net.sourceforge.stripes.controller.*;
import net.sourceforge.stripes.integration.spring.*;
import net.sourceforge.stripes.util.bean.BeanUtil;
@Intercepts ({LifecycleStage.BindingAndValidation}) ❶
public class LoadObjectInterceptor extends
SpringInterceptorSupport
implements Interceptor{
    HibernateTemplate hibernateTemplate;
    private Logger log=Logger.getLogger
(LoadObjectInterceptor.class);
    public Resolution intercept (ExecutionContext ctx) throws
Exception{❷
        Method handler=ctx.getHandler ();
        LoadBean loadProperty=handler.getAnnotation (LoadBean.class); ❸
        if (loadProperty!=null&&loadProperty.value () !=""
&&loadProperty.value () !=null) {❹
            String propertyName=loadProperty.value ();
            String idName=propertyName+".id";
            HttpServletRequest request=ctx.getActionBeanContext
().getRequest (); ❺
            String idValue=request.getParameter (idName);
            Class<?>propertyClass=
```

```

        BeanUtil.getPropertyType (propertyName, ctx.getActionBean () ) ;
⑥
        if (idValue != null && idValue != "") {
            Object o = hibernateTemplate.get (propertyClass,
                Integer.valueOf (idValue) ) ; ⑦
            BeanUtil.setPropertyValue (propertyName,
                ctx.getActionBean () ,
                propertyClass.cast (o) ) ; ⑧
        }
        Resolution resolution = ctx.proceed () ; ⑨
        return resolution;
    }
    @SpringBean ("hibernateTemplate") ⑩
    public void injectHibernateTemplate (HibernateTemplate
aHibernateTemplate) {
        this.hibernateTemplate = aHibernateTemplate;
    }
}

```

① Stripes 已经知道这个类要成为一个 `Interceptor`，因为在 `web.xml` 中已经列出了相关的配置。不过，它的 `@Intercepts` 标注还告诉 Stripes，这个特殊的拦截器类将拦截哪个生命周期阶段。

② `intercept ()` 方法就是在我们感兴趣的生命周期阶段将要调用的方法。在这个方法中，我们想加载数据。需要做的第一件事是找到 Spring 已经选择的事件处理器，以便可以查找我们的 `@LoadBean` 标注。Spring 让这些操作变得很简单，通过 `ExecutionContext.get-Handler ()` 方法就可以返回代表事件处理器的 `Method` 对象。

③ Java 5 的标注机制可以很好地对反射（`reflection`）提供扩展支持，我们只要简单地调用 `Method.getAnnotation ()`，就可以找到事件处理器方法中的 `@LoadBean` 标注（如果确实存在）。

❷ 如果没有找到请求类型的标注，`getAnnotation()` 方法就返回 `null`，所以 `Interceptor` 要检查是否找到了 `@LoadBean` 标注，以及它的值是否足以让 `Hibernate` 能够继续加载数据。`LoadBean.value()` 方法返回原来附加到事件处理器标注上的参数。在例14-21中，将 `propertyName` 的取值设置为了 `album`。

没有谈及的一个命名规范是，我们一直将数据 `bean` 的 `id` 属性命名为 `id`。从 `Stripes` 构建请求参数的方法来看，这意味着只要简单地将字符串 `".id"` 附加到通过 `@LoadBean` 标注找到的 `bean` 名称的后面，就能够构造出包含想要加载的 `bean` 的 `ID` 属性值的参数。在例14-21中，相关的请求参数是 `album.id`。

❸ 我们已经知道了想要加载的请求参数的名称，这样就能够请求参数中查找这个参数，这是通过 `ExecutionContext.getActionBeanContext().getRequest()` 来完成的。

❹ 在拦截器使用 `Spring` 的 `HibernateTemplate` 加载数据对象之前，还需要知道试图加载的对象是什么类型的。`Stripes` 提供了一个名为 `BeanUtil` 的实用工具类来处理 `bean`，以便简化这种与 `bean` 交互的操作。`BeanUtil.getPropertyType()` 返回一个 `Class` 对象，准确地告诉我们需要知道的一切。

⑦装备好了我们想要加载对象的ID和类型，就可以让Hibernate完成加载了。只需要将这些参数传递给HibernateTemplate.get () 方法，即可加载我们想要的数据库。注意，这个Interceptor完全是通用的，它不需要特定类型的DAO，能够在任何在使用这些框架的应用程序中，处理任何绑定到Hibernate映射对象上的数据bean。

⑧BeanUtil也提供了一个setProperty () 方法，可以为数据对象的属性进行赋值。在这里需要将数据库返回的数据对象的类型转换为前面已经准备好的类型，否则调用ActionBean上的setter方法将会失败。

⑨ExecutionContext.proceed () 方法返回一个Resolution实例，将Stripes指向下一件要做的事的链接。

⑩就像ActionBean中的一样，从SpringInterceptorSupport继承的类能够通过Spring的依赖注入来实例化。在这个例子中，我们从Spring中获取HibernateTemplate类。

现在需要让Stripes知道它应该运行刚才创建的LoadObjectInterceptor。为了提供Spring bean注入，我们已经在web.xml中配置好了SpringInterceptor。将LoadObjectInterceptor放在Interceptor.Classes init-param的SpringInterceptor和BeforeAfterMethodInterceptor条目之间，如例14-23所示。

例14-23：在web.xml中将我们的Interceptor添加到StripesFilter

```
.....
<filter>
<display-name>Stripes Filter</display-name>
<filter-name>StripesFilter</filter-name>
<filter-class>
net.sourceforge.stripes.controller.StripesFilter
</filter-class>
<init-param>
<param-name>ActionResolver.PackageFilters</param-name>
<param-value>com.oreilly.*</param-value>
</init-param>
<init-param>
<param-name>ActionResolver.UrlFilters</param-name>
<param-value>WEB-INF/classes</param-value>
</init-param>
<init-param>
<param-name>Interceptor.Classes</param-name>
<param-value>
net.sourceforge.stripes.integration.spring.SpringInterceptor,
com.oreilly.hh.web.LoadObjectInterceptor,
net.sourceforge.stripes.controller.BeforeAfterMethodInterceptor
</param-value>
</init-param>
</filter>
.....
```

在前面的第13章中，DAO对象可以继承HibernateDAOSupport，但我们这里的LoadObject-Interceptor不能这样做。为了使用依赖注入，它现在需要继承Stripes的SpringInterceptorSupport。为此，我们在LoadObjectInterceptor中添加了一个injectHibernateTemplate（）方法，还要在applicationContext.xml中用id hibernateTemplate定义一个HibernateTemplate bean，如例14-24所示。

例14-24: 在applicationContext.xml中为我们的Interceptor注入 HibernateTemplate

```
.....
    <bean
id="hibernateTemplate"class="org.springframework.orm.hibernate3.
    HibernateTemplate">
    <property name="sessionFactory">
    <ref bean="sessionFactory"/>
    </property>
    <property name="cacheQueries">
    <value>true</value>
    </property>
    </bean>
.....
```

再次运行ant compile, Tomcat应该重新加载context, 这时应用程序就应该可以正确处理前面我们无法解决的专辑评论的问题了。在调用AlbumActionBean的save事件处理器以前, 会先调用LoadObjectInterceptor.intercept () 方法。因为在AlbumActionBean.save () 方法上有一个@LoadBean标注, LoadObjectInterceptor就会尝试用请求参数album.id所代表的id来加载Album对象。如果这种方法正常运行, 当Stripes开始调用Album上的setter方法时, 关联数据应该就已经在Album中了。所以, 再调用persist () 方法时, 关联数据就不会丢失了。

这个例子让人感兴趣的部分是它演示了如何扩展Stripes的功能, 这是非常简洁的—所有功能都在幕后运行, 用简单、紧凑的标注来触发它们。

我们已经演示了一种集成Hibernate和Stripes的方法。Stripes带来的实用的功能、先进的编码技术、适度的扩展点，这些让它成为一个使用起来令人感到愉快的Web框架。大多数Web应用程序都需要某种持久化机制，Hibernate的ORM功能与Stripes的自动对象生成机制的组合成就了一种功能更为强大的工具。编写项目不再只是追求可以用什么组件，而是讲究怎么让集成可以更光滑，怎么能让各个组件更好地彼此适应（尤其是用Spring集成各组件）。

你可以将本章研究的一些技术应用到其他Java项目。使用拦截器来检查正在处理的类中保存的元数据标注，这种技术对于为我们的应用程序中增加安全保护之类的功能特别有用。

[1] <http://stripesframework.org/display/stripes/Lifecycles+Etc>.

附录A Hibernate类型

按照数据与持久化服务关系的不同，而对两种不同的数据进行了基本的区分：实体和值。

实体有它自己独立的存在，而不考虑当前在Java虚拟机中是否有任何对象引用了它。通过查询可以从数据库中检索回实体，它们必须由应用程序显式地保存和删除（如果已经建立了级联关系，对父实体的保存或删除动作也会触发它的子对象的保存或删除。但从父实体的角度来看，这种级联仍然是显式的）。

值只是保存为实体的持久化状态的一部分。它们没有自己的独立存在。值可以是原始类型、集合或者用户自定义的类型。因为它们完全从属于赖以存在的实体，所以它们不能被独立地加上版本信息，也不能被多个实体或集合共享。

注意，某个特定的Java对象既可能是实体，也可能是值。区别在于它的设计方式，以及它是如何提供给持久化服务的。原始Java类型总是值。

基本值类型

这里只是简单地列举了一些有关内建类型的信息，演示了它们如何将Java类关联到SQL的字段类型。我们提供了数据库之间存在着的差异的例子，但不打算列举每种差异。有关权威性的细节描述，可以查看org.hibernate.dialect中所有数据库方言实现的源代码（查找register-ColumnType（）调用）。

Hibernate的基本类型可以大致分为：

简单数字和Boolean类型

这些类型都对应于Java的原始类型，可以代表数字、字符、Boolean值或者其封装类型，它们要映射到适当的SQL字段类型（基于使用的SQL方言）。这些类型是：boolean、byte、character、double、float、integer、long、short、true_false以及yes_no。最后两种类型是Boolean值在数据库中的另外两种表示形式，true_false使用"T"和"F"值，而yes_no则使用"Y"和"N"值。

字符串类型

Hibernate类型string可以将java.lang.String映射到与SQL方言相应的字符串字段类型（通常是VARCHAR，或者是Oracle的VARCHAR2）。

日期类型

Hibernate使用date、time以及timestamp，将java.util.Date（及其子类）映射到相应的SQL类型（例如DATE、TIME、TIMESTAMP）。timestamp实现使用的是Java环境中的当前时间；除了使用dbtimestamp，你也可以使用数据库对当前时间的符号表示方法。如果你比较喜欢使用更方便的java.util.Calendar类，在编写自己的代码时也不需要将它和Date类型的值来回转换；你可以用calendar（这个类型将日期和时间保存为TIMESTAMP）或calendar_date（这个类型只接受日期部分，映射为DATE字段类型）直接进行映射。

任意精度的数字类型

Hibernate的big_decimal类型提供从java.math.BigDecimal到相应的SQL类型之间的映射（通常是NUMERIC，但Oracle使用NUMBER）。Hibernate的big_integer提供对java.math.BigInteger的映射（通常映射到BIGINT，但Informix称之为INT8，Oracle则再次使用NUMBER）。

本地化值

Hibernate类型locale、timezone以及currency保存为字符串（VARCHAR或VARCHAR2），并被映射到java.util包中的Locale、TimeZone以及Currency类。Locale和Currency的实例用它们的ISO代码进行保存，而TimeZone则用它的ID属性进行保存。

Class名称

Hibernate的class类型将java.lang.Class的实例映射为它的完全限定的名称，保存在字符串类型的字段中（VARCHAR，或者Oracle中的VARCHAR2）。

字节数组

binary类型将字节数组（byte array）映射为对应的SQL二进制类型。可序列化对象serializable类型用于将可序列化的Java类型映射到对应的SQL二进制类型。这是一种后备类型，当一个对象没有更合适的特定持久化映射时（而且也不想为它实现一个UserType自定义映射，参见下一节），就可以尝试使用这种类型。该类型映射到的SQL类型与binary的映射类型相同，稍后将加以介绍。

JDBC大型数据对象

blob和clob类型为java.sql包中的Blob和Clob类提供映射。如果你正在处理真正的大型数据对象，最好是将这些属性声明为Blob或Clob，即便这样会在数据对象中需要增加一个显式的JDBC依赖。通过Hibernate可以方便地利用JDBC的功能来延迟加载属性值，只有在需要的时候才加载这些大型数据对象。

如果你不担心数据的体积太庞大，就可以不必使用这种直接的JDBC接口，只要将属性类型声明为String或byte[]，再将它用text或binary进行映射。这些类型分别对应于SQL类型的CLOB和VARBINARY

（在Oracle中是RAW，在PostgreSQL中则是BYTEA）。当加载对象时，也会立即将这些值加载到各个属性中。

自定义值类型

除了将对象映射为实体，也可以创建自定义的类，将它们映射为数据库中其他实体的值，而不能自己独立存在。实现这种映射，简单的话，只要改变现有类型的映射方式就可以了（可能你想使用一种不同的字段类型或表示方式）；复杂的话，就需要将一个值划分到多个字段中。

虽然在映射文档中可以在个别的基础上，一个个地来实现映射，但为了遵循尽可能避免代码重复的原则，应该将要在多处使用的类型封装到一个真正可重用的类中。自定义的类可以实现 `org.hibernate.UserType` 或 `org.hibernate.CompositeUserType` 接口。第6章对这种技术进行了介绍。

用这种方法可以对Java 5的enum（枚举）类型（以及旧版本的Java中，手工编码的类型安全的枚举模式的实例）进行映射。可以用一种单独的、可重用的自定义类型映射来支持所有的枚举类型，如第6章所述。

“任意”类型映射

最后介绍一种非常自由的映射。本质上，这种类型的映射可以将引用交替映射到其他多种映射实体。它需要提供两个字段，第一个字段包含每个引用被映射到的表的名称，另一个字段则提供关注的特定实体在那个表中的ID。

在这种松散的映射关系中，不能指定任何外键约束。其实，现实中很少需要这种类型的映射。可能需要使用这种映射的一种情况是，如果你想维护一个审计日志，它可能包含多种实际的对象。参考手册中也提及Web应用程序的会话数据也是另一种潜在的使用情况，但是这样的程序似乎不可能是一种结构良好的应用程序。

所有类型

以下表格列举了org.hibernate.types包中支持的所有类型，以及在映射文档中应该使用的类型名称、映射值在保存时可能使用的最常见的SQL类型、有关类型作用的相关注释。对于许多映射情况，前面已经做了更详细的介绍。除了由其他所有类型实现的Type接口，为了节省页面空间，每种类型名称后面的“类型”（Type）一词就省略掉了。

类型	类型名称	SQL类型	注释
AbstractBinary (或许也可以指 byte/binary/array?)	N/A	N/A	封装用于将字节流绑定到VARBINARY-风格的字段类型的代码
AbstractCharArray	N/A	N/A	封装用于将字符流绑定到VARCHAR-风格的字段类型的代码
AbstractComponent (接口)	N/A	N/A	让Component类型可以保存集合、级联等
Abstract	N/A	N/A	内建类型使用的抽象框架
Any	any	N/A	支持“any”类型映射
Array	array	N/A	将Java数组映射为持久化集合
Association (接口)	N/A	N/A	支持实体间的关联
Bag	bag	N/A	用bag语义对集合进行映射
BigDecimal	big_decimal	NUMERIC	在Oracle中，对应的SQL类型是NUMBER
BigInteger	big_integer	BIGINT	在Oracle中，对应的SQL类型是NUMBER；Informix则使用INT8
Binary	binary	VARBINARY	字节数组的基本类型，非延迟加载（详细参阅本附录）
Blob	blob	BLOB	链接到JDBC，支持延迟加载的字节数组
Boolean	boolean	BIT	基本原始类型
Byte	byte	TINYINT	基本原始类型
CalendarDate	calendar_date	DATE	映射Calendar，忽略时间
Calendar	calendar	TIMESTAMP	映射Calendar，包括时间
Character	character	CHAR	一种基本的原始类型
CharacterArray	N/A	VARCHAR	映射 Character[] 属性
CharArray	N/A	VARCHAR	映射 char[] 属性

(续)

类型	类型名称	SQL 类型	注释
CharBoolean	N/A	CHAR	用于实现yes_no和true_false类型的抽象框架
Class	class	VARCHAR或 VARCHAR2	用于存储类名称的基本类型
Clob	clob	CLOB	链接到JDBC，支持延迟加载的字符数组
Collection	N/A	N/A	支持所有的持久化集合类型
Component	component	N/A	将包含的值类的各属性映射到一组字段
CompositeCustom	N/A	N/A	调整CompositeUserType的实现，以支持一定的Type接口
Currency	currency	VARCHAR或 VARCHAR2	为Currency存储其ISO 代码
Custom	N/A	N/A	调整UserType的实现，以支持一定的Type接口
Date	date	DATE	基本原始类型
DbTimestamp	dbtimestamp	TIMESTAMP	基本原始类型，使用数据库表示的“now”
Discriminator（接口）	N/A	N/A	为用于实现鉴别器（discriminator）属性的类型标识其接口 （选择正确的映射子类）
Double	double	DOUBLE	基本原始类型
EmbeddedComponent	composite - element	N/A	在映射内声明特定的ComponentType
Entity	N/A	N/A	表示到另一个实体的引用
Float	float	FLOAT	基本原始类型
Identifier（接口）	id	N/A	为用于保存实体的标识符的类型标记其接口
IdentifierBag	idbag	N/A	用bag语义映射一个Collection及其标识符字段

Immutable	N/A	N/A	恒定类型（immutable type）的抽象超类；扩展NullableType
Integer	integer	INTEGER	基本原始类型
List	list	N/A	映射Java的List
Literal（接口）	N/A	N/A	为用于保存SQL字面值（literal）的类型而标识其接口
Locale	locale	VARCHAR或 VARCHAR2	为locale（本地化）保存其ISO代码
Long	long	LONG	基本原始类型
ManyToOne	many-to-one	N/A	实体之间的一种关联
Map	map	N/A	映射Java的Map
Meta	meta-type	N/A	为使用any的多态映射保存其鉴别器（discriminator）的值
Mutable	N/A	N/A	非恒定类型（mutable type）的抽象超类
Nullable	N/A	N/A	简单的、可以为null的字段类型的抽象超类
OneToOne	one-to-one	N/A	实体之间的一种关联
OrderedMap	N/A	N/A	MapType 的扩展，以保留SQL 排序
OrderedSet	N/A	N/A	SetType 的扩展，以保留SQL 排序
Primitive	N/A	N/A	用于映射原始Java类型的抽象框架；扩展了ImmutableType
Serializable	serializable	Binary (see JDBC Large Objects earlier)	可序列化类没有更好的映射选择时，可以用这种映射
Set	set	N/A	映射Java的Set
Short	short	SMALLINT	基本原始类型
SortedMap	N/A	N/A	MapType的扩展，以利用Java的有序集合
SortedSet	N/A	N/A	SetType的扩展，以利用Java的有序集合

（续）

类型	类型名称	SQL类型	注释
String	string	VARCHAR or VARCHAR2	基本原始类型
Text	text	CLOB	将CLOB字段以非延迟方式加载到一个String类型的属性中（参见上述说明）
Time	time	TIME	基本原始类型
TimeZone	timezone	VARCHAR or VARCHAR2	保存时区ID
Timestamp	timestamp	TIMESTAMP	基本原始类型，使用JVM表示的“now”
TrueFalse	true_false	CHAR	将Boolean值保存为“T”或者“F”
Type（接口）	N/A	N/A	所有类型的超级接口（superinterface）
Version（接口）	N/A	N/A	为版本戳（version stamp）而扩展Type
WrapperBinary	N/A	N/A	这个类型好像还没有定义好
YesNo	yes_no	CHAR	将Boolean值保存为“Y”或“N”

还有一个TypeFactory类，在为给定的需要而构建正确的Type实现时，可以用它来提供帮助，例如，在映射文档中解析一个类型名称时。阅读一下它的源代码也很有趣。

附录B Criteria API

Criteria查询首先使用`createCriteria()`方法从Session中获取一个Criteria对象，并标明执行查询的对象（主要的类，也就是数据表）。之后再通过下面介绍的各种工厂方法为Criteria附加上约束条件、投影以及排序，这样它就可以成为一个功能非常强大而方便的查询接口了。

Criterion工厂

Restrictions可以作为创建Criterion实例的工厂，用于限制从条件查询中返回的对象（记录行）。Restrictions定义了一组静态方法，通过调用这些方法并传递一定的参数，就可以方便地创建在Hibernate中使用的标准Criterion实现。这些查询条件用于决定在查询结果中最终包含哪些来自数据库的持久化对象。下表总结了Restrictions工厂方法可以提供的选择。

方法	参数	目的
allEq	Map properties	让多个属性的取值为特定值的快捷方式。 Map 对象中的键是需要加以限制的属性的名称，而 Map 中的相应值则是每个属性必须等于的目标取值（如果某个实体要包含在查询结果中的话）。返回的 Criterion 可以确保每个指定的属性都具有相应的取值
and	Criterion lhs, Criterion rhs	创建一个组合 Criterion ，只有当它的两个 Criterion 代表的条件都满足时，整个条件才满足。也可以参阅 conjunction()
between	String property, Object low, Object high	要求指定属性的取值应该落在参数 low 和 high 指定的值的范围内
conjunction	无	创建一个 Conjunction 对象，可以用它来创建任意数目的“and”关系的查询条件。只需要简单地调用它的 add() 方法，

(续)

方法	参数	目的
		并提供需要检查的每个Criterion实例。当且仅当Conjunction中的每个子查询组件都为true时，Conjunction才为true。与使用and()方法来手工构建查询条件的树状结构相比，Conjunction更方便一些。Criteria接口的add()方法看起来表明它内部也包含了一个Conjunction
disjunction	无	创建一个Disjunction 对象，可以用它来创建任意数目的“or”关系的查询条件。只需要简单地调用它的add()方法，并提供需要检查的每个Criterion实例。如果Disjunction中的任何一个子查询组件为true，Disjunction就为true。与使用or()方法来手工构建查询条件的树状结构相比，Disjunction更方便一些。相关示例可以参阅例8-10
eq	String property, Object value	要求指定属性具有特定的值
eqProperty	String property1, String property2	要求两个指定属性的取值相同
ge	String property, Object value	要求指定属性的值大于或等于指定的值
geProperty	String property1, String property2	要求第一个指定的属性的取值大于或等于第二个属性的取值
gt	String property, Object value	要求指定属性的取值大于特定的值
gtProperty	String property1, String property2	要求第一个指定属性的取值大于第二个属性的取值
idEq	Object value	要求标识符（identifier）属性等于指定的值
ilike	String property, Object value	大小写不敏感的“like”运算符，参见“like”
ilike	String property, String value, MatchMode mode	大小写不敏感的“like”运算符，供不喜欢复杂的“like”语法，只需要匹配一定的子字符串的用户使用。MatchMode是一个类型安全的枚举值，可选取的值为START、END、ANYWHERE以及 EXACT。这个方法会根据mode属性指定的匹配模式来调整value的语法结构，接着再像两个参数的ilike()方法那样进行处理

(续)

方法	参数	目的
in	String property, Collection values	要求指定的属性取值可以选取集合中包含的任意值。与手工建立eq()查询条件的disjunction()方法相比, 这个方法更方便
in	String property, Object[] values	要求指定的属性取值可以选取数组中包含的任意值。与手工建立eq()查询条件的disjunction()方法相比, 这个方法更方便
isEmpty	String property	要求指定的集合属性是空的(成员个数为0)
isNotEmpty	String property	要求指定的集合属性至少要有有一个元素
isNotNull	String property	要求指定的属性不能包含null值
isNull	String property	要求指定的属性为null
le	String property, Object value	要求指定的属性小于或等于特定的值。参见例8-3
leProperty	String property1, String property2	要求指定的第1个属性小于或等于第2个属性
like	String property, Object value	要求指定的属性“like”于特定的值(从SQL like运算符的意义上来说, 它支持简单的子字符串匹配)。参见例8-8和例8-16
like	String property, String value, MatchMode mode	为不喜欢复杂的“like”运算符语法, 只想匹配一定的子字符串的人提供的“like”运算符。更多细节请参阅ilike()方法
lt	String property, Object value	要求指定的属性小于特定的值
ltProperty	String property1, String property2	要求指定的第1个属性小于第2个属性
naturalId	None	支持通过<natural-id>映射的多列“自然业务键”(natural business key)的选择
ne	String property, Object value	要求指定的属性不能取特定的值
neProperty	String property1, String property2	要求指定的两个属性具有不同的值
not	Criterion expression	对给定的Criterion求反(如果Criterion匹配, 则为false, 反之亦然)

(续)

方法	参数	目的
or	Criterion lhs, Criterion rhs	构建一个复合Criterion, 只要它的任意一个子Criterion匹配, 则该复合条件就成功。参见disjunction()
sizeEq	String property, int size	要求指定的集合属性具有特定数量的子元素
sizeGe	String property, int size	要求指定的集合属性至少包含一个特定的元素
sizeGt	String property, int size	要求指定的集合属性的成员个数大于特定的值
sizeLe	String property, int size	要求指定的集合属性的成员个数不超过特定的值
sizeLt	String property, int size	要求指定的集合属性的成员个数少于特定的值
sizeNe	String property, int size	要求指定的集合属性中互不相同的成员的个数超过size
sqlRestriction	String sql	采用底层数据库系统的原生SQL方言来表达限制条件。虽然这个功能强大, 但要小心因此而失去可移植性的意义
sqlRestriction	String sql, Object[] values, Type[] types	采用底层数据库系统的原生SQL及多个JDBC参数来表达限制条件。虽然这个功能强大, 但要小心因此而失去可移植性的意义
sqlRestriction	String sql, Object value, Type type	采用底层数据库系统的原生SQL及JDBC参数来表达限制条件。虽然这个功能强大, 但要小心因此而失去可移植性的意义

当为sqlRestriction () 方法指定查询语句文本时, 查询语句中出现的任何"{alias}"字符串都将由执行查询涉及的数据表的实际别名所取代。

这些方法中的多数都以Criterion实例作为参数, 可以按照你需要的任意复杂程度来构建复合条件查询树。通过conjunction () 和

`disjunction()` 返回的对象可以方便地添加新的查询条件，多次调用 `add()` 方法可以添加任意多个条件。不过，如果查询足够复杂的话，用HQL进行查询可能更容易表达和理解。还有小量的查询种类用这种API无法提供支持，所以不可能总能避免使用HQL。但是这种情况会越来越少，大多数这类基本的查询在应用程序开发的整个过程中都会用到，而用这种API来表达简单的查询也非常自然和容易，同时也让Java代码变得更加可读和简洁，并在编译时就检查代码是否正确。

Projection工厂

Hibernate提供的`org.hibernate.criterion.Projections`可以作为创建投影实例的工厂，用投影可以缩小从条件查询中返回的属性（列）的个数，以及计算聚合（`aggregate`）值。可以调用投影类提供的静态方法，并使用提供的参数来方便地创建Hibernate中使用的标准投影实现。这些投影用于缩小查询结果中属性的范围、对属性进行分组或转换。以下列举了Hibernate提供的一些可用选项。

方法	参数	目的
alias	Projection projection, String alias	为投影指派一个别名（名称），以便在Criteria查询的其他地方可以引用该投影（例如分组、排序等）
avg	String property	计算指定属性的平均值
count	String property	计算在结果中指定属性出现的次数
countDistinct	String property	计算在结果中取值不相同的指定属性出现的次数
distinct	Projection projection	让投影查询只返回具有惟一值的记录（去掉取值重复的记录）
groupByProperty	String property	按指定的属性对查询结果进行分组（可与聚合值计算一起使用）。参见例8-15
id	None	将对象的标识符作为投影的一个元素（不论标识符属性的名称是什么）
max	String property	计算指定属性的最大值。参见例8-15
min	String property	计算指定属性的最小值
projectionList	None	创建一个新的投影列表，用于请求多个投影值。参见例8-14
property	String property	在投影中包括指定的属性。参见例8-13
rowCount	None	计算返回结果中的记录行的个数。参见8-15
sqlGroupProjection	String sql, String groupBy, String[] columnAliases, Type[] types	支持使用数据库特定的SQL代码来执行投影；groupBy可以包含一个SQL “GROUP BY” 子句。这里需要告诉Hibernate投影中返回列的别名和类型
sqlProjection	String sql, String[] columnAliases, Type[] types	支持使用数据库特定的SQL代码来执行投影；可以使用上面的groupBy()投影来执行分组。这里需要告诉Hibernate投影中返回列的别名和类型
sum	String property	计算指定属性值的代数和

Order工厂

可以让Hibernate（它再让底层的数据库系统）对查询结果按一定的顺序进行排序。org.hibernate.criterion.Order类代表这些排序请求，提

供了两个静态工厂方法用于创建实例，再将这些实例附加到条件查询中。在获得Order实例后，如果需要对结果排序不区分字母大小写，则可以调用Order实例上的非静态方法ignoreCase（）。

方法	参数	目的
asc	String property	按照指定属性的升序排列对结果进行排序。参见例8-6
desc	String property	按照指定属性的降序排列对结果进行排序

Property工厂

在目前介绍的方法中，都是先按照感兴趣的内容来创建条件查询、投影或者排序实例，再将查询涉及的属性名称作为参数传递给相关方法。Criteria API也支持用与上述相反的方向来进行处理，先从属性开始，再调用一个方法来构建基于该属性的查询或投影。

org.hibernate.criterion.Property是一个创建Property实例的工厂，如果你喜欢后面这种构建查询的方法，就可以使用这个类。Property定义了一个静态的forName（）方法，调用它就可以创建一个代表特定属性的实例。在获得实例以后，就可以再调用该实例提供的方法来创建基于它代表的属性上的条件查询、投影以及排序。本书这里只列举一些经常使用到的方法，省略介绍的方法主要与离线查询（detached criteria）和子查询有关，这些主题已经超出本书的范围。当你需要了解它们时，可以看看《Java Persistence with Hibernate》中的"Advanced query

options"（高级查询选项）这一章，或是在线参考文档中的"Detached queries and subqueries"（[\[1\]](#)）（离线查询和子查询）这一部分。

方法	参数	目的
asc	None	创建一个Order，根据指定属性值的升序顺序对查询结果进行排序
avg	None	创建一个Projection，返回属性值的平均值
between	Object min, Object max	创建一个Criterion，要求属性值介于min 和 max之间
count	None	创建一个Projection，返回属性值在查询结果中出现的次数 ^①

(续)

方法	参数	目的
desc	None	创建一个Order，根据指定属性值的降序顺序对查询结果进行排序
eq	Object value	创建一个Criterion，要求属性值等于提供的值 ^②
eqProperty	Property other	创建一个Criterion，要求属性值等于另一个由other代表的属性值
eqProperty	String property	创建一个Criterion，要求属性值等于另一个其名称由property指定的属性值
ge	Object value	创建一个Criterion，要求属性值大于或等于指定的值 ^②
geProperty	Property other	创建一个Criterion，要求属性值大于或等于另一个由other代表的属性值
geProperty	String property	创建一个Criterion，要求属性值大于或等于另一个其名称由property指定的属性值
getProperty	String property	从当前属性中提取指定名称的复合属性元素（假定当前属性是一个复合属性）。返回另一个Property实例
group	None	创建一个Projection，要求按当前属性值对查询结果进行分组
gt	Object value	创建一个Criterion，要求属性值大于指定的值 ^②
gtProperty	Property other	创建一个Criterion，要求属性值大于另一个由other代表的属性值
gtProperty	String property	创建一个Criterion，要求属性值大于另一个其名称由property指定的属性值
in	Collection values	创建一个Criterion，要求属性值包含于提供的集合中
in	Object[] values	创建一个Criterion，要求属性值应该等于提供的数组中的某个元素
isEmpty	None	创建一个Criterion，要求属性应该为空（包含0个元素的集合）
isNotEmpty	None	创建一个Criterion，要求属性应该为至少包含一个元素的集合
isNotNull	None	创建一个Criterion，要求属性值不能为null
isNull	None	创建一个Criterion，要求属性值为null
le	Object value	创建一个Criterion，要求属性值小于或等于指定的值 ^②

(续)

方法	参数	目的
leProperty	Property other	创建一个Criterion，要求属性值小于或等于另一个由other代表的属性值
leProperty	String property	创建一个Criterion，要求属性值小于或等于另一个其名称由property指定的属性值
like	Object value	创建一个Criterion，要求属性值“like”于指定的值（从SQL like运算符的意义来说，它支持简单的子字符串匹配） ^②
like	String value, MatchMode mode	供不喜欢复杂的“like”语法，只需要匹配一定的子字符串的用户使用的“like”方法。MatchMode是一个类型安全的枚举值，可选取的值为START、END、ANYWHERE以及EXACT。这个方法会根据mode属性指定的匹配模式来调整value的语法结构，接着再像一个参数的like()方法那样进行处理 ^②
lt	Object value	创建一个Criterion，要求属性值小于指定的值 ^②
ltProperty	Property other	创建一个Criterion，要求属性值小于另一个由other代表的属性值
ltProperty	String property	创建一个Criterion，要求属性值小于另一个其名称由property指定的属性值
max	None	创建一个Projection，返回当前属性的最大值
min	None	创建一个Projection，返回当前属性的最小值
ne	Object value	创建一个Criterion，要求属性值不能取指定的值 ^②
neProperty	Property other	创建一个Criterion，要求属性值不能等于另一个由other代表的属性值
neProperty	String property	创建一个Criterion，要求属性值不能等于另一个其名称由property指定的属性值

注：① 该方法的返回类（CountProjection）有一个setDistinct()方法，如果需要统计不同属性值的个数，则可以调用这个方法。

② 该方法的返回类（SimpleExpression）有一个ignoreCase()方法，如果需要进行不区分大小写的比较，则可以调用这个方法。

[1]

[http://www.hibernate.org/hib_docs/v3/reference/en/html/querycriteria.html#querycriteria-detachedqueries.](http://www.hibernate.org/hib_docs/v3/reference/en/html/querycriteria.html#querycriteria-detachedqueries)

附录C Hibernate SQL方言

掌握流利的SQL方言

Hibernate封装了对很多商业（[\[1\]](#)）和免费的关系数据库的支持。虽然不进行这些配置，大多数功能也都可以正常运行，但是将hibernate.dialect配置属性设置成正确的org.hibernate.dialect.Dialect子类具有一定的重要性，尤其是在使用诸如native或sequence主键生成以及会话锁定的功能时。如果你指定一种方言，Hibernate将为一些配置参数使用合理的默认值，为你省去了手工单独指定它们的麻烦。

数据库系统	相应的hibernate.dialect设置
Caché 2007.1	org.hibernate.dialect.Cache71Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
Derby	org.hibernate.dialect.DerbyDialect
Firebird	org.hibernate.dialect.FirebirdDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
H2	org.hibernate.dialect.H2Dialect
HSQLDB	org.hibernate.dialect.HSQLDialect
Informix	org.hibernate.dialect.InformixDialect
Ingres	org.hibernate.dialect.IngresDialect
Interbase	org.hibernate.dialect.InterbaseDialect

(续)

数据库系统	相应的hibernate.dialect设置
JDataStore	org.hibernate.dialect.JDataStore
Mckoi SQL	org.hibernate.dialect.MckoiDialect
Mimer SQL	org.hibernate.dialect.MimerSQLDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
MySQL (versions prior to 5.x)	org.hibernate.dialect.MySQLDialect
MySQL (version 5.x and later)	org.hibernate.dialect.MySQL5Dialect
MySQL (prior to 5.x, using InnoDB tables)	org.hibernate.dialect.MySQLInnoDBDialect
MySQL (prior to 5.x, using MyISAM tables)	org.hibernate.dialect.MySQLMyISAMDialect
MySQL (version 5.x, using InnoDB tables)	org.hibernate.dialect.MySQL5InnoDBDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 8i	org.hibernate.dialect.Oracle8iDialect
Oracle 9i or 10g	org.hibernate.dialect.Oracle9Dialect
Oracle 10g only (use of ANSI join syntax)	org.hibernate.dialect.Oracle10gDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
Progress	org.hibernate.dialect.ProgressDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Sybase (or MS SQL Server)	org.hibernate.dialect.SybaseDialect
Sybase 11.9.2	org.hibernate.dialect.Sybase11Dialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Teradata	org.hibernate.dialect.TeradataDialect
TimesTen 5.1	org.hibernate.dialect.TimesTenDialect
Unisys 2200 RDMS	org.hibernate.dialect.RDMSOS2200Dialect

如果以上没有你需要的目标数据库，可以检查一下最新发布的Hibernate是否已经提供了相关的支持。Hibernate参考文档的"SQL Dialects"（[\[2\]](#)）部分列出了能够支持的大部分SQL方言。如果还是不

行，可以再看看是否能够找到其他第三方对相关数据库的支持，或者考虑自己动手开发！

[1] 我从来就没有指望可以再次利用到缓存中的东西，还是把软件健壮性的问题留给Java吧.....

[2] <http://www.hibernate.org/hib-docs/v3/reference/en/html/session-configuration.html#configuration-optional-dialects>.

附录D Spring事务支持

使用Spring Framework的事务标注

在具体的类或者接口上可以添加`Transactional`标注，为类或方法增加事务管理。如果用`Transactional`对一个类进行标注，其设置将会应用到类中定义的每个方法。如果用`Transactional`对一个方法进行标注，事务设置将只应用到某个单独的方法。如果对类和方法都是应用了`Transactional`标注，对方法进行的标注比对类进行的标注具有更高的优先权。

通过`Transactional`标注，可以控制事务的隔离级别、超时时间、传播（`propagation`）设置以及会导致事务回滚的一组异常。例如，如果我们希望总是用可序列化的隔离级别创建一个新的事务，在一分钟之内没有完成或发生了`NumberFormatException`异常，就回滚事务，就可以编写类似例D-1所示的代码。

例D-1：更多的事务配置控制选项

```
@Transactional (readOnly=false,
propagation=Propagation.REQUIRES_NEW,
isolation=Isolation.SERIALIZABLE,
rollbackFor={NumberFormatException.class},
timeout=60)
public abstract void run () ;
```

如果使用Transactional标注，需要小心选择标注放置的位置。如果先使用了Spring中的代理，或者准备要研究一下Spring中为Aspect-Oriented Programming（AOP）提供的引人注目的支持，你需要注意尽可能避免把Transactional标注放到接口上，还需要注意要将Transactional标注放在非public的方法上。在这个示例中，因为我们没有使用任何Spring的AOP功能，所以就把Transactional标注放在了一个接口上。如果你正在为一个系统调试错误，其中省略了Transactional标注，这时需要做的第一件事情就是验证标注过的方法是否是public的。如果使用AspectJ或早期的Spring AOP功能，在使用代理或AOP时，总会遇到些令人混淆的问题。如果决定使用Spring的AOP功能，就应该记得务必将@Transactional标注放在具体的类上。

注意：如果对这些术语不熟悉，看看下面表格的解释会有帮助。

Transactional标注属性

表D-1列举了Transactional标注的属性。

表D-1：Transactional标注的属性

表D-1：Transactional标注的属性

标注属性	类型	描述
isolation	Isolation	事务隔离设置。后面的表D-3提供了可用选项的更多细节
noRollbackFor	Class[]	一组不会导致事务回滚的异常类（或类名）。
noRollbackForClassName	String[]	可以指定一组完全限定的类名或Class对象
propagation	Propagation	事务传播配置。有关这一配置的更多细节，可以参见表D-2
readOnly	boolean	指定事务是只读的，还是可读写的
rollbackFor	Class[]	一组将会导致事务回滚的异常类（或类名）。
rollbackForClassName	String[]	RuntimeExceptions的默认行为就是会触发回滚，而对于checked exception则通常不会触发回滚。如果抛出了checked exception，并希望因此而触发回滚，就需要把这个异常添加到rollbackFor属性。可以指定一组完全限定的类名或Class对象
timeout	int	事务超时需要经过的秒数。如果将这个属性设置为-1，则表示事务永远不会超时（无限的超时时间）

事务传播

propagation属性用于告诉Spring，操作是否需要一个新的事务、嵌套事务或者还是在现有的事务中进行操作。表D-2列出了propagation属性的有效取值。

事务的传播行为依赖于事务提供者。如果你正在使用的是JTA，一定要仔细阅读文档，看看是否支持嵌套事务。

表D-2: Propagation枚举值

值	描述
Propagation.REQUIRED	使用现有的事务；如果没有任何事务，就创建一个新事务（这是默认值）
Propagation.SUPPORTS	将参与到现有的事务中；如果不存在已有的事务，也不会创建新事务
Propagation.MANDATORY	需要提供一个现有的事务；如果不存在现有的，则抛出异常
Propagation.REQUIRES_NEW	为这个方法创建一个新事务；如果存在，则挂起当前事务
Propagation.NOT_SUPPORTED	标注方法中的持久化操作将不在事务中执行。如果在已经有的事务中调用该方法，则该事务会被挂起
Propagation.NEVER	如果在一个事务当中调用该方法，则抛出异常
Propagation.NESTED	如果在一个现有的事务中调用该方法，则在嵌套事务中执行该标注方法

事务隔离

isolation属性可以控制在事务中需要什么锁，以及数据库中当前正在执行的事务和状态对事务的影响。表D-3列举了isolation属性的有效取值。

表D-3: Isolation枚举值

值	描述
Isolation.DEFAULT	使用底层数据库的默认隔离级别
Isolation.SERIALIZABLE	提供最高级别的隔离级别，不能读取“脏”数据，不能重复读取，也不能幻像读取（phantom read）。当使用这种隔离级别时，即使读操作也需要获得锁。在使用这种隔离级别（包括高于READ_UNCOMMITTED以上的隔离级别）时，应该小心避免两个活动事务之间的死锁（deadlock）冲突
Isolation.REPEATABLE_READ	不能读取“脏”数据和重复读取。但可能幻读
Isolation.READ_COMMITTED	不能读取“脏”数据。但可能重复读和幻读
Isolation.READ_UNCOMMITTED	最低级别的隔离级别，一个事务可以看到另一个事务未提交的修改。脏读、重复读以及幻读在这个级别内都有可能发生

在真实世界的系统中使用SERIALIZABLE隔离级别也经常是不现实的，因为对数据库的访问不会全部都“顺序依次”发生。通常使用的是REPEATABLE_READ或READ_COMMITTED隔离级别，并构建一些检测死锁和在失败后尝试重新操作的逻辑代码。大型多用户应用程序（例如Web网站）在使用SERIALIZABLE时需要当心事务死锁，如果使用的是高于READ_UNCOMMITTED的任何级别，就应该确保在Transactional标注中为timeout属性定义了一个有限的超时时间值。事务隔离的具体行为会依赖于正在使用的数据库，例如，MySQL的InnoDB存储引擎对事务隔离级别的解释就与Oracle、SQL Server、Derby或HSQLDB中的版本存在一些差异。

使用JTA事务管理器

在第13章的示例中，我们使用的是HibernateTransactionManager，因为它用起来比较直观和简单。如果需要使用JTA来参与分布式事务，或者处理跨越多种技术（JDBC+JMS）的事务，那么在applicationContext.xml中可以用其他东西来取代transactionManager，如例D-2所示。

例D-2：配置JTA事务管理器

```
<!--enable the configuration of transactional behavior based on
annotations-->
<tx:annotation-driven transaction-manager="transactionManager"/
>
<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager"
/>
```

附录E 参考资料

为了更深入地挖掘Hibernate的能力，以及使用它的其他方法，还有许多研究学习的渠道。下面就介绍一些好的习学资源，挑选些最适合你需要的方法，确定好你的学习风格和时间安排。

在线手册

本书中涉及的开发工具都提供了不错的在线文档。如果你想了解一下这些开发包的基本用法，在线参考手册可以给你提供有关如何完成特定任务的详细介绍。在每个开发包主页的显著位置上都可以找到Documentation链接。

图书

由Christian Bauer和Gavin King编著的《Java Persistence with Hibernate》（Manning Publications）更加完整而详细地讨论了Hibernate。这两位专家作为Hibernate的创作者，他们非常熟悉Hibernate的细节，不过有时也运用了些相当深奥的专业的数据库概念。

为了掌握些专业知识，你也可以阅读一下George Reese写的《Java Database Best Practices》（O'Reilly），或者至少读一下我们在本书第4章中提到的相关章节（有在线文档可用）（[\[1\]](#)）。要深入研究Apache Maven的使用，可以阅读来自Sonatype的《Maven: The Definitive Guide》。为了更多地了解Apache Ant，可以看看Jesse E.Tilly和Eric M.Burke写的《Ant: The Definitive Guide》（O'Reilly）（[\[2\]](#)），或者是读一下最近由Steve Loughran和Erik Hatcher写的《Ant in Action》（Manning）。

有关Spring Framework的更多信息，可以选择参考由Bruce Tate和Justin Gehtland写的《Spring: A Developer's Notebook》（O'Reilly），或者是参考由Rod Johnson、Juergen Hoeller、Alef Arendsen、Thomas Risberg以及Colin Sampleneau合著的《Professional Java Development with the Spring Framework》（WROX）。虽然本书的作者们可能倾向于多推荐O'Reilly的书目，不过WROX的那本书是由创建和维护Spring Framework的那帮人编写的。

源代码

和任何开源项目一样，最终真正的主宰者还是源代码。通过阅读源代码可以学到不少东西，而不仅仅是你直接关注的任务。当然，作为Java开发包，根据源代码生成的JavaDoc也具有相当的参考价值。如

果你在用Eclipse，就可以下载项目的源代码发布版本，再告诉Eclipse每个JAR库的源代码目录树在什么地方。这样做的价值是，在编写完成类、方法名称、参数时，Eclipse会提示完整的JavaDoc信息，通过F3快捷键也可以打开每个程序元素的源代码，就好像这些开源代码是你自己的项目一样。我们发现这是一种非常具有方向性和辅助学习的好办法。

如果使用Maven，并运行eclipse: eclipse构建目标，则可以通过以下命令来下载源代码和JavaDoc：

```
mvn eclipse: eclipse-DdownloadSources=true  
-DdownloadJavadocs=true
```

下载每个依赖的所有源代码和JavaDoc包可能需要花费不少时间。在本地Maven仓库下载好源代码和JavaDoc以后，就可以在Eclipse IDE中查看依赖库的源代码和JavaDoc了。目前广泛使用的开源开发库的绝大部分（但并非全部）都提供了源代码和JavaDoc工件。

处理更新版本的工具包

对于印刷出版的图书来说，软件的不断变化总是个问题。因此，当你在使用本书涉及的所有这些软件的新版本时，可能会遇到些麻烦。这本书的新版所讨论的示例基于以下特定版本的各种工具：

Ant 1. 7.0 （有时会遇到麻烦）

Eclipse 3. 3.1.1

Geronimo JTA 1. 1 implementation

Hibernate 3. 2.5

Hibernate Annotations 3. 3.0.ga

Hibernate Commons Annotations 3. 3.0.ga

Hibernate Tools 3. 2.0 beta 9a （很多地方）

Hibernate Tools 3. 2.0.GA （第11章）

HSQLDB 1. 8.0.7

Log4J 1. 2.14 （一般不会出现问題）

Maven 2. 0.8

MySQL 5. 0.21

Spring Framework 2. 5

Stripes 1. 4.3

有些工具的更新版本可能会改变它们的工作方式，而且这些变化有时还是向后不兼容的。这也是跟上开源项目发展而带来的部分乐趣。我们提供的**Maven**规则应该可以帮助你找到我们要使用的工具的特定版本，这样会对你的学习和试验过程有一定帮助。你也可以检查这些工具包的发行说明，以决定在什么时候和如何移植到最新、功能最强大的版本。

你还可以在本书网站（[\[3\]](#)）的勘误页面上看看是否也有其他人遇到了与你同样的问题，或许会找到解决的办法。如果没有，也请你尽可能提交自己的发现，与别人一起分享！

Hibernate和**HSQLDB**都有各自的在线支持论坛，为应付各种不兼容的变化提供了不少有价值的建议，也可以学到这些工具其他方面更为复杂的东西。

重在参与

开放源代码带来的最好的一件事情就是你可以使用软件，还可以参与到开发社区中来。如果你使用**Hibernate**、**Spring**、**Ant**、**Maven**或者**Stripes**，那么应该订阅它们的用户邮件列表，以保持不断了解其更新和新版本情况。如果你需要对源代码进行定制，最好将你的修改贡献出来，所有这些项目都有非常成熟的开发社区，以便集成你对代码所做的修改。在对技术掌握精深以后，可以发挥一下主观精

神，为开发人员的邮件列表贡献一些文档补丁。不必对参与有所顾虑；只有每个人都发挥主观能动性，开源事业才会茁壮成长。在为这些项目做贡献时，你不必向任何人申请许可，所有你需要做的就是鼓起勇气、开始参与。

要了解有关Hibernate社区的更多详情，并着手参与这一项目，可以访问Hibernate的官方网站<http://www.hibernate.org>。你可以订阅为用户和开发人员提供的Hibernate邮件列表，或者是按照主页上的提示来看看它的支持论坛：<http://www.hibernate.org/20.html>。Hibernate团队的博客也非常活跃，其网址是<http://blog.hibernate.org/>。将这个博客添加到你的RSS阅读器中，用这个好办法可以与相关的技术保持同步（例如，Seam和Hibernate Shards），Gavin和其他人在这方面的开发都很活跃。

Spring Framework与一个叫做Spring Source（<http://www.springsource.com>）的公司有关系，这个公司雇用了很多志愿者为Spring项目做贡献。有关Spring Framework社区的更多信息，可以访问<http://www.springframework.org>。如果你想订阅邮件列表或者为Spring Framework贡献源代码，则应该看看开发页面上的信息，网址是<http://www.springframework.org/development>。如果你真的对Spring Framework怀有激情，那么应该考虑参加Spring Source的年度会议：The Spring Experience。有关Spring培训的信息可以在Spring Source公

司的网站上找到，有关The Spring Experience的信息则可以访问该会议的网站<http://www.thespringexperience.com/>。

Apache Ant和Apache Maven是Apache Software Foundation (ASF) 的两个顶级项目 (top-level project)。Apache Software Foundation是一个大型的开发人员社区，相关信息可以访问其官方网站<http://www.apache.org>。Apache这个大型组织负责维护一些世界上使用最广泛的项目，例如经常使用的Apache Web服务器和Tomcat应用程序服务器，以及Jakarta Commons开发库。有关Apache Maven开发社区的详情，可以访问其项目网站<http://maven.apache.org>。有关Apache Ant开发社区的更多详情，也可以访问Ant的项目网站<http://ant.apache.org>。Ant和Maven都是非常活跃的社区，其用户和开发人员的邮件列表的信息量都非常大。

Stripes的主页是<http://www.stripesframework.org>，在这个网站上可以找到相关的教程、参考文档以及JavaDoc。要订阅这个项目在SourceForge上的邮件列表，可以访问https://source-forge.net/mail/?group_id=145476。对于那些Java语言的极客 (geek) 来说，浏览Stripes的源代码应该是种有趣的体验，因为这个项目的开发者们显然精通并使用了目前Java的最新功能。我强烈推荐下载Stripes的源代码包，随意看看，权当只是为了兴趣而已。

[1] <http://www.oreilly.com/catalog/javadtbp/chapter/ch02.pdf>.

[2] <http://www.sonatype.com/book/index.html>.

[3] <http://www.oreilly.com/catalog/9780596517724/>.

作者简介

James Elliott是Berbee公司的一位资深软件工程师，有近20年的系统开发专业经验。他对计算机的专注和激情早在职业生涯之前就已经显现，当他在Mexico City上中学时就得到了一台标着"Apple II"的古怪设备，从此就开始了在计算机领域的驰骋。

在工作环境尚未十分方便之前，Jim就已经开始了面向对象的开发。他热衷于建造高质量的工具和框架来简化其他开发人员（而不只是他自己）的工作，喜欢研究如何有效地使用Java才能有助于解决实际问题，尤其是当这一语言趋于成熟以后。

在经历了环游世界般的童年时代以后，Jim在位于纽约州的Rensselaer Polytechnic Institute大学取得了学士学位，在University of Wisconsin-Madison取得了硕士学位，在那期间，他也在贝尔实验室（位于Murray Hill, C语言和UNIX的诞生地）从事一些有趣的工作。虽然在通过博士资格考试以后，Jim马上欣然接受了现实世界的诱惑，但是能在Madison找到有趣的工作让他很高兴。他目前和他的搭档Joe Buberger住在Madison。

Ryan Fowler是Berbee公司的一位软件工程师。他的编程生涯开始于Apple II机器上的BASIC语言，那时他还在Michigan州Grand Rapids

的St.Stephen School读小学。一段时间后，他又回到Michigan州Alma的Alma College计算机科学系写代码，并取得了学士学位。Ryan喜欢滑雪、航海，有时，也弹弹吉他来打发时间。Ryan和他的妻子居住在Wisconsin的Madison。Tim O'Brien是一位Emacs信徒，最近又转向了Apple Macintosh计算机。Tim早在20世纪80年代初就开始在TRS-80上编写程序，后来在University of Virginia学习电子工程。作为独立的技术人员，Tim经常与Grassroots Technologies合作；他最近开发了一种混搭体系结构，可以为金融信息、客户保护、汽车以及教育出版业的各种客户端提供服务。在闲暇时间，Tim喜欢睡觉、写作以及参与开源文档项目。Tim目前和妻子Susan、女儿Josephine居住在Illinois的Evanston。

封面介绍

本书封面上的动物叫做刺猬（**hedgehog**），是一种刺猬亚科（**Erinaceinae**）小动物。目前已经发现了5属16种刺猬，广泛分布在欧洲、亚洲、非洲和新西兰。其中包括：四趾刺猬（非洲撒哈拉以南）、长耳刺猬（中亚）、沙漠刺猬（亚洲和中东）、光腹刺猬（印度）。不同类的刺猬体型大小各不相同，一般长5到12英寸，重15到40盎司。作为驯养的刺猬通常是四趾刺猬或非洲侏儒刺猬，个头比它们的欧洲同类要小很多，目前已经在很多国家成为流行的宠物。刺猬一词最早出现于15世纪中期——“**hedge**（树篱）”，因为它的根长在地下；而“**hog**”（猪獾）则是因为它长着像猪一样的口鼻。刺猬的其他名称还有**urchin**、**hedgepig**和**fuzzy-pig**。

刺猬最与众不同的特点就是它的棘刺，除了面部、腿和腹部以外，全身都长满了刺。当受到惊吓时，它会卷缩成球状，全身棘刺竖立，让入侵者无从下手。这些刺坚硬而中空，由角质层发育而成，非常坚固。与豪猪的刺不同，刺猬的刺不能脱落，除非在称为“蜕皮”的过程中它的体刺才会脱落。

刺猬的主要食物是一些小型无脊椎动物，例如青蛙、毛毛虫以及蚯蚓。一些刺猬能够抵抗许多毒素，所以它们可以吃蜜蜂、黄蜂甚至是毒蛇。刺猬是在夜间活动的动物，而白天躲在草丛中岩石下面或地

洞中睡大觉。尽管所有的刺猬都可以冬眠，但也不总是这样，因为冬眠要由各种因素来决定，比如地点、温度以及食物的充足程度。在英国，每年11月5日的焰火之夜（**Bonfire Night**）的庆祝会给刺猬带来很大的危险，因为这些小动物经常会躲在用于燃放篝火的木料堆中睡觉。野生动物保护组织向公众警告，为了保护正在冬眠的刺猬，在点火以前一定要检查一下他们的木料堆中是否有这些可爱的小动物。

封面图片来自于J.G.Wood的Animate Creation。